



ASP.NET MVC 4 框架揭秘

◎蒋金楠 著



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

ASP.NET MVC 4 框架揭秘

蒋金楠 著

電子工業出版社

Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

针对最新版本的 ASP.NET MVC 4, 深入剖析底层框架从请求接收到响应回复的整个处理流程(包括 URL 路由、Controller 的激活、Model 元数据的解析、Model 的绑定、Model 的验证、Action 的执行、View 的呈现和 ASP.NET Web API 等), 并在此基础上指导读者如何通过对 ASP.NET MVC 框架本身的扩展解决应用开发中的实际问题。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有, 侵权必究。

图书在版编目(CIP)数据

ASP.NET MVC 4 框架揭秘 / 蒋金楠著. —北京: 电子工业出版社, 2013.1
ISBN 978-7-121-19049-0

I. ①A… II. ①蒋… III. ①网页制作工具—程序设计 IV. ①TP393.092

中国版本图书馆 CIP 数据核字(2012)第 281603 号

策划编辑: 张春雨

责任编辑: 葛 娜

印 刷: 北京东光印刷厂

装 订: 三河市鹏成印业有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1092 1/16 印张: 37 字数: 855 千字

印 次: 2013 年 1 月第 1 次印刷

印 数: 3000 册 定价: 89.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

前言

ASP.NET MVC 是一个建立在 ASP.NET 平台上基于 MVC 模式的 Web 开发框架,它提供了一种与传统 Web Forms 完全不同的 Web 应用开发方式。ASP.NET Web Forms 借鉴了 Windows Forms 基于控件和事件注册的编程模式,使 Web 应用的开发变得简单而快捷,但是它却使开发人员与 Web 的本质渐行渐远。ASP.NET MVC 是一种回归,它使开发人员可以真正地面向 Web 进行编程,我们面对的不再是拖拉到 Web 页面的控件,而是整个 HTTP 请求和响应的流程。

这不是一本 ASP.NET MVC 入门书籍

我个人觉得掌握 ASP.NET MVC 具有三个层次。了解基本的编程模式,掌握 Controller 和 View 的定义方式,知道路由如何注册,以及验证规则如何定义,此为第一层次。第二层次要求我们对 ASP.NET MVC 框架本身从请求接收到响应回复的整个流程具有一个清晰的认识,包括请求如何被路由、目标 Controller 如何被激活、Model 元数据如何被解析、Action 方法如何被执行、View 如何呈现等。ASP.NET MVC 本身是一个极具可扩展的开发框架,合理利用其扩展性可以解决很多开发中的实际问题,而掌握 ASP.NET MVC 的最高层次就是凭着对框架本身的运行机制的了解准确地找到相应的扩展点,并创建相应的扩展来解决我们遇到的问题。本书不是一本 ASP.NET MVC 入门书籍,而是让处于第一层次的读者快速进入第二和第三层次的书。

这是一本讲述 ASP.NET MVC 框架本质的书

很多.NET 开发人员都在抱怨微软开发技术过快的更新频率让他们无所适从。其实他们看到的只是单纯的版本升级而已,一些本质的东西一直是“稳定”的。微软推出.NET 战略已经十多年了,CLR 却只有四个版本而已。最新版本的 ASP.NET 虽然表面上已经看不到太多最初的影子,但是整个请求处理的管道一直未曾改变。对于一项开发技术,只要我们了解了它最根本的一些东西,就不应该惧怕其高频率的版本更替,而应该热烈拥抱它。本书力求

将关于 ASP.NET MVC 框架最根本的东西带给大家，而不是罗列一些简单的编程技巧。

这是一本实用的书

可能有人觉得这本剖析 ASP.NET MVC 框架运行原理的书没有什么“实际”的意义，因为我们每天的日常工作就是编程，知道了 ASP.NET MVC 从请求接收到响应回复之间整个处理流程并不会对我们的工作造成实质性的改变。其实这种想法是极端错误的，因为我们编写的程序最终是在 ASP.NET MVC 框架上运行的，程序的高效性决定于它是否能够最大限度地“迎合”框架的运行机制，所以了解 ASP.NET MVC 框架的运行原理有利于我们写出高质量的程序。

我个人将基于 ASP.NET MVC 的编程分为两类，即“面向业务编程”和“面向框架编程”。前者根据具体的业务逻辑定义 Controller 和设计 View，这是大部分 Web 开发人员的主要工作；后者则是为整个 Web 应用搭建一个框架，让最终的开发人员只需要关注具体的业务逻辑，而让框架来完成所有与业务无关的部分。对于后者，我们可以充分利用 ASP.NET MVC 的扩展性，通过自定义的扩展将非业务的功能自动“注入”到业务逻辑的处理流程中，这样不仅可以提高开发效率，而且还能提高开发质量。本书在剖析 ASP.NET MVC 框架运行机制过程中几乎列出了其所有的扩展点，并且通过实例演示的形式提供了很多实用的扩展。

可以将本书视为一本“架构设计”的书

在我的周围存在这样的一些人，他们以刚毕业一两年的毕业生为主，他们大都工作勤奋、聪明好学，手中经常捧着 GoF 的《设计模式》，总是希望将书中的设计模式应用到具体项目之中，或者希望通过项目的实践来印证他们在书本上的设计模式，但是理论和实践之间的距离总让他们感到困惑。

要从真实的项目或者产品中学习“实用”的软件架构设计知识，先得确定目标项目或者产品中采用的架构思想和设计模式是正确的，而我们参与的很多项目其实被“架构”得一塌糊涂。对于像 ASP.NET 这样的产品，其基础架构能够在很长一段时间内保持不变，本身就证明了应用在上面的架构设计的正确性，它们不正是我们学习架构设计最好的素材吗？本书对 ASP.NET MVC 框架的运行机制进行了深入剖析，实际上是将 ASP.NET MVC 的整个设计展示在读者面前，读者朋友们也许可以将本书作为一本“架构设计”的书来读。

本书的写作特点

我想本书的读者可能很多都读过我的《WCF 全面解析》，虽然内容不同，本书却可能看成是它的延续，因为它们基本上采用了相同的写作手法。总地来讲，我基本上采用“原理讲

述、代码分析和实例证明”这个模式来介绍某个技术要点，对于一个具体的知识点，我不仅会告诉读者“是什么”，还会告诉读者“为什么”，以及“如何证明是这样”。除此之外，如果某个知识点在真实的项目开发中具有“实用”价值，我一般会给出一个相关的实例演示。

本书具有一个与其他中文原创或者翻译书籍截然不同的特点，那就是几乎所有的术语都采用英文，比如 Controller、Model 和 View 在本书中并没有翻译成中文“控制器”、“模型”和“视图”。因为我认为很多术语其实很难找到一个语义完全等同的中文词组或短语与之对应，对于习惯了英文作为“开发语言”的读者来说，强行翻译其实是不必要的。

这不是一本纯理论的书，而是一本“实证型”的书，在书中提供了 110 个可供单独下载的实例演示。这些实例在本书中具有不同的作用，有的是为了探测和证明对应的论点，有的是为了演示某种使用的编程技巧，有的直接就是一个完整的案例。

本书读者

我们说《ASP.NET MVC 4 框架揭秘》不是一本 ASP.NET MVC 入门书籍，主要是因为本书在第 1 章并没有提供一个“Hello World”实例，关注重点主要落在 ASP.NET MVC 框架本身的运行机制上面，但是并不是说本书的读者需要预先对 ASP.NET MVC 具有多深入的认识才行。如果读者对 ASP.NET MVC 基本的编程模式具有一定的了解，读懂这本书是完全没有问题的。对于从未接触过 ASP.NET MVC 的 .NET 开发人员，可以通过官方网站 (<http://www.asp.net/mvc>) 来学习 ASP.NET MVC。

本书结构

第 1 章 ASP.NET + MVC

ASP.NET 和 MVC，分别代表了 ASP.NET MVC 的技术平台和设计思想。本章对 MVC 模式及其变体比如 MVP 和 Model 2 等作了概括性介绍，同时对 ASP.NET 的管道式设计，以及与各种版本的 IIS 之间的交互机制进行了全面讲述。为了让读者对 ASP.NET MVC 框架的运行机制具有一个大概的了解，本章按照其原理创建了一个“迷你版”的 ASP.NET MVC。

第 2 章 URL 路由

ASP.NET MVC 借助于 URL 路由系统实现了 URL 模式与目标 Controller 和 Action 的映射，以及内嵌于 URL 的参数传递。基于 URL 路由的编程主要体现在路由映射的注册和基于注册路由的 URL 生成上面，本章对这两个方面作了非常详细的介绍。URL 路由最终是借助于自定义的 HttpModule (UrlRoutingModule) 实现的，它利用动态注册 HttpHandler 映射的

方式提供针对 URL 路由的实现，这是本章着重讲述的重点。

第 3 章 Controller 的激活

本章对以 `ControllerFactory` 为核心的 Controller 激活系统，以及通过 `DefaultControllerFactory` 提供的 Controller 默认激活机制进行了详细介绍。以 IoC 的方式激活 Controller 在实际的 Web 应用开发中具有重要的意义，本章以较多的篇幅讲述了如何将不同的 IoC 框架（Unity 和 Ninject）应用到 ASP.NET MVC 的 Controller 激活系统中。具体来说，我们以实例演示的方式讲述了三种不同的实现方式，包括自定义 `ControllerFactory`、`ControllerActivator` 和 `DependencyResolver`。

第 4 章 Model 元数据的解析

Model 元数据是针对数据类型的一种描述信息，ASP.NET MVC 提供了基于数据注解特性的声明式 Model 元数据定义方式，本章对所有与此相关的数据注解特性，以及它们对 Model 元数据的影响进行了全面的介绍。ASP.NET MVC 利用 Model 元数据实现了模板化的 HTML 生成方式，本章重点讲述了如何为具体的数据类型定义编辑和显示模板，以及定义的模板在调用 `HtmlHelper/HtmlHelper<TModel>` 的模板方法过程中是如何控制最终生成的 HTML 的。本章的最后关注于以 `ModelMetadataProvider` 为核心的 Model 元数据提供机制，以及如何通过自定义 `ModelMetadataProvider` 实现对 Model 元数据提供机制的定制。

第 5 章 Model 的绑定

ASP.NET MVC 的 Model 绑定旨在为目标 Action 方法提供参数列表。`ParameterDescriptor` 为 Model 绑定提供了相关的元数据信息，本章以介绍 `ParameterDescriptor` 以及相关的 `ControllerDescriptor` 和 `ActionDescriptor` 作为开篇。Model 绑定所需的最终数据通过 `ValueProvider` 来提供，本章接下来会对实现在各种不同 `ValueProvider` 中的数据值提供机制，以及以 `ValueProviderFactory` 为核心的 `ValueProvider` 提供机制进行全面而深入的介绍。本章的最后部分着重介绍以 `ModelBinder` 为核心的 Model 绑定系统，以及实现在 `DefaultModelBinder` 中的默认 Model 绑定机制。

第 6 章 Model 的验证

Action 方法在执行之前需要通过 Model 验证机制确保提供参数的有效性。本章会着重讲述以 `ModelValidator` 为核心的 Model 验证系统，以及通过 `ModelValidatorProvider` 实现的 `ModelValidator` 提供机制。Model 验证是伴随着 Model 绑定进行的，具体执行流程的介绍也

包含在本章之中。ASP.NET MVC 利用 `ValidationAttribute` 特性为 Model 验证提供了一种声明式编程方式，其背后的实现机制是本章重要讲述的内容。jQuery 验证框架被默认用于客户端验证，jQuery 验证的编程方式，以及与 ASP.NET MVC 验证系统的协作方式会在本章的最后一部分予以介绍。

第 7 章 Action 的执行

针对请求的处理最终体现在对目标 Action 方法的执行上面。Action 方法可以以同步或者异步的方式执行，所以本章以介绍两种不同的异步 Action 编程模式作为开篇；此外，同步与异步的差异体现在整个请求的处理过程中，`MvcHandler`、`Controller`、`ActionInvoker`、`ControllerDescriptor` 和 `ActionDescriptor` 等都具有同步和异步的版本，本章会对它们作一个系统的比较。Action 的执行还伴随着筛选器的执行，在本章的最后对四种筛选器的作用和执行流程进行单独介绍。

第 8 章 View 的呈现

`ActionResult` 作为执行 Action 返回的结果，实现了对请求的最终响应，本章介绍了所有预定义的 `ActionResult` 分别是如何完成针对请求的响应的。作为最重要的 `ActionResult`，`ViewResult` 将整个预定义的 View 呈现出来，而它背后是一套完整的 View 引擎，View 引擎的运行机制，以及与 `ViewResult` 的协作方式是本章介绍的一个重点。ASP.NET MVC 默认提供了 ASPX 和 Razor 这两种原生 View 引擎的支持，针对 Razor 引擎的深入剖析被放在本章的最后一部分。

第 9 章 ASP.NET Web API

ASP.NET Web API 使我们可以很容易地定义 REST 服务，本章会提供 Web API 基本编程模式的介绍。ASP.NET Web API 采用了与 ASP.NET MVC 独立但类似的执行管道，对整个管道从请求接收到响应回复的整个流程的介绍是本章的重点，包括 `HttpController` 的激活与执行、Action 的选择、Model 元数据的解析、Action 参数的绑定与执行等。

第 10 章 案例实践

本章提供了一个名为 Video Mall（简称 VM）的在线电子商务购物网站来模拟 ASP.NET MVC 在真实项目中的应用。VM 以 SQL Server 作为数据存储，并采用 Entity Framework 作为 ORM 框架进行数据存取。VM 利用了在前面章节中定义的一系列扩展，同时还涉及了一些架构思想和涉及模式，比如模块化设计、IoC、AOP 以及 Repository 等。

关于作者

蒋金楠（网名 Artech）现就职于某知名软件公司担任高级软件顾问。连续 5 届微软 MVP（最有价值专家），同时也是少数的双料 MVP（Solutions Architecture + Connected System）之一。国内较早接触 WCF 的人之一，2007 年 2 月起在个人博客（<http://www.cnblogs.com/artech>）上发表超过两百篇深入介绍 WCF 的文章，成为了目前国内 WCF 在线资料的主要来源。

致谢

本书得以出版，需要感谢本书的编辑张春雨先生和葛娜小姐，你们的专业水准和责任心是为本书提供的质量保证，期待着与你们第三度合作的机会。此外，最需要感谢的是我的老婆徐妍妍，只有我知道你在本书提交给出版社之前所作的校对工作有多么重要。

本书支持

本书针对最新版本的 ASP.NET MVC，同时涉及太多底层实现的内容，所以大部分内容是找不到任何现成参考资料的，这些内容大都来自于作者对源码的分析和试验的证明。本书的最初版本是根据 ASP.NET MVC 4 Beta 撰写的，差不多快写完的时候微软发布了 ASP.NET MVC 4 RC，然后我根据 RC 对原来的内容作了不小的改动。在 ASP.NET MVC 4 正式推出之后，我第一时间联系到了 Scott Guthrie，从他们团队得到了一份正式版与 RC 之间变化的列表，并据此又作了一些修改。这些因素加上我本人能力的限制，都可能造成一些无法避免的错误或者偏差，如果读者在阅读过程中发现了任何问题，希望能够反馈给我。如果读者遇到任何 ASP.NET MVC 或者是 WCF 的问题，也欢迎与我通过以下的方式进行交流。

- 作者博客：<http://www.cnblogs.com/artech>
- 作者微博：<http://www.weibo.com/artech>
- 电子邮箱：jiangjinnan@gmail.com

本书每一章节都会提供一系列实例演示，读者可以根据编号（比如 S101、S202 等）从下载的源代码压缩包中找到对应的实例。本书的附录给出了所有源代码可供下载的所有的实例演示的列表和相关描述。

- 源代码下载地址：<http://files.cnblogs.com/artech/asp.net.mvc.4.samples.rar>

目 录

第 1 章 ASP.NET + MVC	1
1.1 传统 MVC 模式	2
1.1.1 自治视图	2
1.1.2 什么是 MVC 模式	3
1.2 MVC 的变体	4
1.2.1 MVP	4
1.2.2 Model 2	12
1.2.3 ASP.NETMVC 与 Model 2	13
1.3 IIS/ASP.NET 管道	14
1.3.1 IIS 5.x 与 ASP.NET	14
1.3.2 IIS 6.0 与 ASP.NET	15
1.3.3 IIS 7.0 与 ASP.NET	17
1.3.4 ASP.NET 管道	20
1.4 ASP.NET MVC 是如何运行的	25
1.4.1 建立在“迷你版”ASP.NET MVC 上的 Web 应用	25
1.4.2 URL 路由	27
1.4.3 Controller 的激活	31
1.4.4 Action 的执行	35
本章小结	39
第 2 章 URL 路由	41
2.1 ASP.NET 路由系统	42
2.1.1 请求 URL 与物理文件的分离	42
2.1.2 实例演示：通过 URL 路由实现请求地址与.aspx 页面的映射（S201）	43
2.1.3 Route 与 RouteTable	46
2.1.4 路由映射	52
2.1.5 根据路由规则生成 URL	59

2.2	ASP.NET MVC 扩展	61
2.2.1	路由映射	61
2.2.2	实例演示：注册路由映射与查看路由信息（S208）	62
2.2.3	缺省 URL 参数	65
2.2.4	基于 Area 的路由映射	67
2.2.5	链接和 URL 的生成	71
2.3	动态 <code>HttpHandler</code> 映射	78
2.3.1	<code>UrlRoutingModule</code>	78
2.3.2	<code>PageRouteHandler</code> 与 <code>MvcRouteHandler</code>	79
2.3.3	ASP.NET 路由系统扩展	80
	本章小结	85
第 3 章	Controller 的激活	86
3.1	总体设计	87
3.1.1	<code>Controller</code>	87
3.1.2	<code>ControllerFactory</code>	92
3.1.3	<code>ControllerBuilder</code>	93
3.1.4	<code>Controller</code> 的激活与 URL 路由	99
3.2	默认实现	101
3.2.1	<code>Controller</code> 类型的解析	102
3.2.2	<code>Controller</code> 类型的缓存	105
3.2.3	<code>Controller</code> 的释放和会话状态行为的控制	106
3.3	IoC 的应用	108
3.3.1	从 <code>Unity</code> 来认识 IoC	108
3.3.2	<code>Controller</code> 与 <code>Model</code> 的分离	110
3.3.3	基于 IoC 的 <code>ControllerFactory</code>	111
3.3.4	基于 IoC 的 <code>ControllerActivator</code>	117
3.3.5	基于 IoC 的 <code>DependencyResolver</code>	119
	本章小结	122
第 4 章	Model 元数据的解析	123
4.1	Model 元数据及其定制	124
4.1.1	Model 元数据层次化结构	124
4.1.2	基本 Model 元数据信息	125
4.1.3	Model 元数据的定制	128
4.1.4	<code>IMetadataAware</code> 接口	142
4.2	Model 元数据与 Model 模板	146

4.2.1 实例演示：通过模板将布尔值显示为 RadioButton (S409)	147
4.2.2 预定义模板	148
4.2.3 DataTypeName 与模板名称	157
4.2.4 模板的获取与执行	160
4.2.5 实例演示：通过定制 Model 元数据和自定义模板 实现预定义列表的呈现 (S412)	164
4.3 Model 元数据的提供机制	172
4.3.1 再谈 ModelMetadata	172
4.3.2 ModelMetadataProvider	176
4.3.3 Model 元数据提供系统的扩展	180
本章小结	182
第 5 章 Model 的绑定	183
5.1 ControllerDescriptor、ActionDescriptor 与 ParameterDescriptor	184
5.1.1 ControllerDescriptor	184
5.1.2 ActionDescriptor	189
5.1.3 ParameterDescriptor	193
5.2 ValueProvider	196
5.2.1 NameValueCollectionValueProvider	197
5.2.2 DictionaryValueProvider	203
5.2.3 ValueProviderFactory	211
5.2.4 ValueProviderFactories	211
5.3 ModelBinder	215
5.3.1 ModelBinder 与 ModelBinderProvider	215
5.3.2 ModelState 与 Model 绑定	223
5.3.3 ModelBindingContext 的创建	227
5.4 Model 绑定的默认实现	228
5.4.1 简单类型	229
5.4.2 复杂类型	232
5.4.3 数组	238
5.4.4 集合	246
5.4.5 字典	248
本章小结	252
第 6 章 Model 的验证	254
6.1 ModelValidator 与 ModelValidatorProvider	255
6.1.1 ModelValidator	255

6.1.2	ModelValidatorProvider	258
6.1.3	ModelValidatorProviders	264
6.2	Model 绑定与验证	269
6.2.1	ModelState	269
6.2.2	验证消息的呈现	272
6.2.3	Model 绑定中的验证	278
6.3	基于数据注解特性的 Model 验证	283
6.3.1	ValidationAttribute 特性	283
6.3.2	DataAnnotationsModelValidator	290
6.3.3	DataAnnotationsModelValidatorProvider	292
6.3.4	将 ValidationAttribute 应用到参数上	295
6.3.5	一种 Model 类型, 多种验证规则	300
6.4	客户端验证	307
6.4.1	jQuery 验证	307
6.4.2	基于 jQuery 的 Model 验证	311
6.4.3	自定义验证	315
	本章小结	318
第 7 章	Action 的执行	320
7.1	异步 Action 的定义	321
7.1.1	基于线程池的请求处理机制	321
7.1.2	两种异步 Action 方法的定义	322
7.1.3	AsyncManager	324
7.2	Action 方法的执行	330
7.2.1	MvcHandler 对请求的处理	330
7.2.2	Controller 的执行	330
7.2.3	ActionInvoker 的执行	331
7.2.4	ControllerDescriptor 的同步与异步	336
7.2.5	ActionDescriptor 的执行	339
7.3	筛选器的执行	345
7.3.1	Filter 及其提供机制	345
7.3.2	AuthorizationFilter	355
7.3.3	ActionFilter	365
7.3.4	ExceptionHandler	371
7.3.5	实例演示: 集成 EntLib 实现自动化异常处理 (S713, S714, S715)	373
7.3.6	ResultFilter	387
	本章小结	388

第 8 章 View 的呈现	390
8.1 ActionResult	391
8.1.1 EmptyResult	391
8.1.2 ContentResult	392
8.1.3 FileResult	398
8.1.4 JavaScriptResult	402
8.1.5 JsonResult	405
8.1.6 HttpStatusCodeResult	408
8.1.7 RedirectResult/RedirectToRouteResult	409
8.2 ViewResult 与 ViewEngine	411
8.2.1 View 引擎中的 View	411
8.2.2 ViewEngine	413
8.2.3 ViewResult 的执行	415
8.3 Razor 引擎	423
8.3.1 View 的编译原理	423
8.3.2 WebViewPage 与 WebViewPage<TModel>	427
8.3.3 RazorView	432
8.3.4 RazorViewEngine	441
本章小结	444
第 9 章 ASP.NET Web API	445
9.1 Web、REST 与 Web API	446
9.1.1 Web 如此简单	446
9.1.2 REST 是什么	447
9.1.3 ASP.NET Web API	450
9.2 服务端管道	458
9.2.1 ASP.NET Web API 管道式设计	459
9.2.2 HttpResponseMessage	461
9.2.3 HttpServer	464
9.2.4 实例演示：自定义 HttpResponseMessage 实现 HTTP 方法重写（S903）	469
9.3 ApiControllerDispatcher	471
9.3.1 ApiController 的激活	472
9.3.2 ApiController 的执行	485
9.3.3 Action 的选择	486
9.3.4 Model 元数据的解析	492
9.3.5 Action 参数绑定	495
9.3.6 Model 验证	508

9.3.7 Action 的执行与结果的响应.....	512
9.4 Web API 的调用和自我寄宿.....	516
9.4.1 HttpClient.....	516
9.4.2 HttpSelfHostServer.....	521
本章小结.....	525
第 10 章 案例实践.....	527
10.1 功能性简介.....	528
10.1.1 商品列表的呈现.....	528
10.1.2 订购商品.....	530
10.1.3 登录与错误页面.....	531
10.2 设计概述.....	532
10.2.1 Controller-Service-Repository.....	532
10.2.2 IoC 的应用.....	536
10.2.3 AOP 的应用.....	539
10.2.4 异常处理.....	545
10.3 编程实现.....	546
10.3.1 数据表的创建.....	546
10.3.2 Repository.....	548
10.3.3 Service.....	552
10.3.4 路由注册和布局.....	555
10.3.5 ProductController.....	558
10.3.6 OrderController.....	565
10.3.7 AccountController.....	571
本章小结.....	574
附录 A 实例列表.....	575

第 1 章 ASP.NET + MVC

ASP.NET MVC 是一个全新的 Web 应用框架。将术语 ASP.NET MVC 拆分开来,即 ASP.NET+MVC,前者代表支撑该应用框架的技术平台,意味着 ASP.NET MVC 和传统的 Web Forms 应用框架一样都是建立在 ASP.NET 平台之上;后者则表示该框架背后的设计思想,意味着 ASP.NET MVC 采用了 MVC 架构模式。

1.1 传统 MVC 模式

对于大部分面向最终用户的应用来说，它们都需要具有一个可视化的 UI 界面与用户进行交互，我们将这个 UI 称为视图（View）。在早期，我们倾向于将所有与 UI 相关的操作糅合在一起，这些操作包括 UI 界面的呈现、用于交互操作的捕捉与响应、业务流程的执行以及对数据的存取，我们将这种设计模式称为自治视图（Autonomous View，AV）。

1.1.1 自治视图

说到自治视图，很多人会感到陌生，但是我们（尤其是 .NET 开发人员）可能经常在采用这种模式来设计我们的应用。Windows Forms 和 ASP.NET Web Forms 虽然分别属于 GUI 和 Web 开发框架，但是它们都采用了事件驱动的开发方式，所有与 UI 相关的逻辑都可以定义在针对视图（Windows Forms 或者 Web Forms）的后台代码（Code Behind）中，并最终注册到视图本身或者视图元素（控件）的相应事件上。

一个典型的人机交互应用具有三个主要的关注点，即数据在可视化界面上的呈现、UI 处理逻辑（用于处理用户交互式操作的逻辑）和业务逻辑。自治视图模式将三者混合在一起，势必会带来如下一些问题：

- 业务逻辑是与 UI 无关的，应该最大限度地被重用。由于业务逻辑定义在自治视图中，相当于完全与视图本身绑定在一起，如果我们能够将 UI 的行为抽象出来，基于抽象化 UI 的处理逻辑也是可以被共享的。但是定义在自治视图中的 UI 处理逻辑完全丧失了重用的可能。
- 业务逻辑具有最强的稳定性，UI 处理逻辑次之，而可视化界面上的呈现最差（比如我们会为了更好地呈现效果来调整 HTML）。如果将具有不同稳定性的元素融为一体，那么具有最差稳定性的元素决定了整体的稳定性，这是“短板理论”在软件设计中的体现。
- 任何涉及 UI 的组件都不易测试。UI 是呈现给人看的，并且用于与人进行交互，用机器来模拟活生生的人来对组件实施自动化测试不是一件容易的事，自治视图严重损害了组件的可测试性。

为了解决自治视图导致的这些问题，我们需要采用关注点分离（Seperation of Concerns, SoC）的方针将可视化界面呈现、UI 处理逻辑和业务逻辑三者分离出来，并且采用合理的交互方式将它们之间的依赖降到最低。将三者“分而治之”，自然也使 UI 逻辑和业务逻辑变得更容易测试，测试驱动设计与开发变成了可能。这里用于进行关注点分离的模式就是 MVC。

1.1.2 什么是 MVC 模式

MVC 的创建者是 Trygve M. H. Reenskau, 他是挪威的计算机专家, 同时也是奥斯陆大学的名誉教授。MVC 是他在 1979 年访问施乐帕克研究中心(Xerox Palo Alto Research Center, Xerox PARC) 期间提出一种主要针对 GUI 应用的软件架构模式。MVC 最初用于 SmallTalk, Trygve 最初对 MVC 的描述记录在 *Applications Programming in Smalltalk-80(TM):How to use Model-View-Controller (MVC)* 这篇论文中, 有兴趣的读者可以通过地址 <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html> 阅读这篇论文。

MVC 体现了关注点分离这一基本的设计方针, 它将构成一个人机交互应用涉及的功能分为 Model、Controller 和 View 三部分, 它们各自具有相应的职责。

- Model 是对应用状态和业务功能的封装, 我们可以将它理解为同时包含数据和行为的领域模型 (Domain Model)。Model 接受 Controller 的请求并完成相应的业务处理, 在状态改变的时候向 View 发出相应的通知。
- View 实现可视化界面的呈现并捕捉最终用户的交互操作 (比如鼠标和键盘操作)。
- View 捕获到用户交互操作后会直接转发给 Controller, 后者完成相应的 UI 逻辑。如果需要涉及业务功能的调用, Controller 会直接调用 Model。在完成 UI 处理之后, Controller 会根据需要控制原 View 或者创建新的 View 对用户交互操作予以响应。

图 1-1 揭示了 MVC 模式下 Model、View 和 Controller 之间的交互。对于传统的 MVC 模式, 很多人认为 Controller 仅仅是 View 和 Model 之间的中介, 实则不然, View 和 Model 存在直接的联系。View 可以直接调用 Model 查询其状态信息。当 Model 状态发生改变的时候, 它也可以直接通知 View。比如在一个提供股票实时价位的应用中, 维护股价信息的 Model 在股价变化的情况下可以直接通知相关的 View 改变其显示信息。

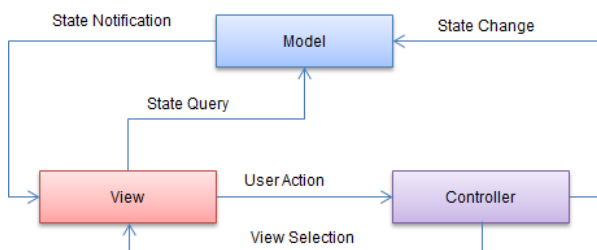


图 1-1 Model-View-Controller 之间的交互

从消息交换模式的角度来讲, Model 针对 View 的状态通知和 View 针对 Controller 的用户交互通知都是单向的, 我们推荐采用事件机制来实现这两种类型的通知。从设计模式的角度来讲就是采用观察者(Observer)模式通过注册/订阅的方式来实现它们, 即 View 作为 Model 的观察者通过注册相应的事件来检测状态的改变, 而 Controller 作为 View 的观察者通过注

册相应的事件来处理用户的交互操作。

我看到很多人将 MVC 和所谓的“三层架构”进行比较，其实两者并没有什么可比性，MVC 更不是分别对应着 UI、业务逻辑和数据存取三个层次，不过两者也不能说完全没有关系。Trygve M. H. Reenskau 当时提出 MVC 的时候是将其作为构建整个 GUI 应用的架构模式，这种情况下的 Model 实际上维护着整个应用的状态并实现了所有的业务逻辑，所以它更多地体现为一个领域模型。而对于多层架构来说（比如我们经常提及的三层架构），MVC 是被当成 UI 呈现层（Presentation Layer）的设计模式，而 Model 则更多地体现为访问业务层的入口（Gateway）。如果采用面向服务的设计，业务功能被定义成相应服务并通过接口（契约）的形式暴露出来，这里的 Model 还可以表示成进行服务调用的代理。

1.2 MVC 的变体

通过采用 MVC 模式，我们可以将可视化 UI 元素的呈现、UI 处理逻辑和业务逻辑分别定义在 View、Controller 和 Model 中，但是对于三者之间的交互，MVC 并没有进行严格的限制。最为典型的就是允许 View 和 Model 绕开 Controller 进行直接交互，View 可以通过调用 Model 获取需要呈现给用户的数据，Model 也可以直接通知 View 让其感知到状态的变化。当我们将 MVC 应用于具体的项目开发中，不论是基于 GUI 的桌面应用还是基于 Web UI 的 Web 应用，如果不对 Model、View 和 Controller 之间的交互进行更为严格的限制，我们编写的程序可能比自治视图更加难以维护。

今天我们将 MVC 视为一种模式（Pattern），但是作为 MVC 最初提出者的 Trygve M. H. Reenskau 却将 MVC 视为一种范例（Paradigm），这可以从它在 *Applications Programming in Smalltalk-80(TM):How to use Model-View-Controller (MVC)* 中对 MVC 的描述看出来：*In the MVC paradigm the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object, each specialized for its task.*

模式和范例的区别在于前者可以直接应用到具体的应用上，而后者则仅提供一些基本的指导方针。在我看来 MVC 是一个很宽泛的概念，任何基于 Model、View 和 Controller 对 UI 应用进行分解的设计都可以成为 MVC。当我们采用 MVC 的思想来设计 UI 应用的时候，应该根据开发框架（比如 Windows Forms、WPF 和 Web Forms）的特点对 Model、View 和 Controller 的界限以及相互之间的交互设置一个更为严格的规则。

在软件设计的发展历程中出现了一些 MVC 的变体（Variation），它们遵循定义在 MVC 中的基本原则，我们现在来简单地讨论一些常用的 MVC 变体。

1.2.1 MVP

MVP 是一种广泛使用的 UI 架构模式，适用于基于事件驱动的应用框架，比如 ASP.NET

Web Forms 和 Windows Forms 应用。MVP 中的 M 和 V 分别对应于 MVC 的 Model 和 View，而 P (Presenter) 则自然代替了 MVC 中的 Controller。但是 MVP 并非仅仅体现在从 Controller 到 Presenter 的转换，更多地体现在 Model、View 和 Presenter 之间的交互上。

MVC 模式中元素之间“混乱”的交互主要体现在允许 View 和 Model 绕开 Controller 进行单独“交流”，这在 MVP 模式中得到了彻底解决。如图 1-2 所示，能够与 Model 直接进行交互的仅限于 Presenter，View 只能通过 Presenter 间接地调用 Model。Model 的独立性在这里得到了真正的体现，它不仅仅与可视化元素的呈现 (View) 无关，与 UI 处理逻辑 (Presenter) 也无关。使用 MVP 的应用是用户驱动的而非 Model 驱动的，所以 Model 不需要主动通知 View 以提醒状态发生了改变。

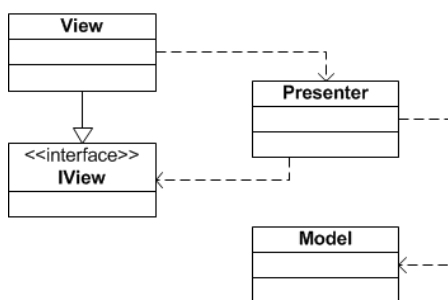


图 1-2 Model-View-Presenter 之间的交互

MVP 不仅仅避免了 View 和 Model 之间的耦合，更进一步地降低了 Presenter 对 View 的依赖。如图 1-2 所示，Presenter 依赖的是一个抽象化的 View，即 View 实现的接口 IView，这带来的最直接的好处就是使定义在 Presenter 中的 UI 处理逻辑变得易于测试。由于 Presenter 对 View 的依赖行为定义在接口 IView 中，我们只需要 Mock 一个实现了该接口的 View 就能对 Presenter 进行测试。

构成 MVP 三要素之间的交互体现在两个方面，即 View/Presenter 和 Presenter/Model。Presenter 和 Model 之间的交互很清晰，仅仅体现在 Presenter 对 Model 的单向调用。而 View 和 Presenter 之间该采用怎样的交互方式是整个 MVP 的核心，MVP 针对关注点分离的初衷能否体现在具体的应用中很大程度上取决于两者之间的交互方式是否正确。按照 View 和 Presenter 之间的交互方式以及 View 本身的职责范围，Martin Folwer 将 MVP 可分为 PV (Passive View) 和 SC (Supervising Controller) 两种模式。

PV 与 SC

解决 View 难以测试的最好的办法就是让它无需测试，如果 View 不需要测试，其先决条件就是让它尽可能不涉及到 UI 处理逻辑，这就是 PV 模式目的所在。顾名思义，PV (Passive View) 是一个被动的 View，包含其中的针对 UI 元素（比如控件）的操作不是由 View 自身主动来控制，而被动地交给 Presenter 来操控。

如果我们纯粹地采用 PV 模式来设计 View，意味着我们需要将 View 中的 UI 元素通过属性的形式暴露出来。具体来说，当我们在为 View 定义接口的时候，需要定义基于 UI 元素的属性使 Presenter 可以对 View 进行细粒度操作，但这并不意味着我们直接将 View 上的控件暴露出来。举个简单的例子，假设我们开发的 HR 系统中具有如图 1-3 所示的一个 Web 页面，我们通过它可以获取某个部门的员工列表。



图 1-3 员工查询页面

现在通过 ASP.NET Web Forms 应用来设计这个页面，我们来讨论一下如果采用 PV 模式，View 的接口该如何定义。对于 Presenter 来说，View 供它操作的控件有两个，一个是包含所有部门列表的 DropDownList，另一个则是显示员工列表的 GridView。在页面加载的时候，Presenter 将部门列表绑定在 DropDownList 上，与此同时包含所有员工的列表被绑定到 GridView。当用户选择某个部门并点击“查询”按钮后，View 将包含筛选部门在内的查询请求转发给 Presenter，后者筛选出相应的员工列表之后将其绑定到 GridView。

如果我们为该 View 定义一个接口 IEmployeeSearchView，我们不能按照所示的代码将上述这两个控件直接以属性的形式暴露出来。针对具体控件类型的数据绑定属于 View 的内部细节（比如说针对部门列表的显示，我们可以选择 DropDownList 也可以选择 ListBox），不能体现在表示用于抽象 View 的接口中。另外，理想情况下定义在 Presenter 中的 UI 处理逻辑应该是与具体的技术平台无关的，如果在接口中涉及控件类型，这无疑将 Presenter 也与具体的技术平台绑定在了一起。

```
public interface IEmployeeSearchView
{
    DropDownList      Departments { get; }
    GridView           Employees { get; }
}
```

正确的接口和实现该接口的 View（一个 Web 页面）应该采用如下的定义方式。Presenter 通过对属性 Departments 和 Employees 赋值进而实现对相应 DropDownList 和 GridView 的数据绑定，通过属性 SelectedDepartment 得到用户选择的筛选部门。为了尽可能让接口只暴露必需的信息，我们特意将对属性的读/写作了控制。

```
public interface IEmployeeSearchView
{
```

```

        IEnumerable<string>      Departments { set; }
        string                   SelectedDepartment { get; }
        IEnumerable<Employee>    Employees { set; }
    }

    public partial class EmployeeSearchView: Page, IEmployeeSearchView
    {
        //其他成员
        public IEnumerable<string> Departments
        {
            set
            {
                this.DropDownListDepartments.DataSource = value;
                this.DropDownListDepartments.DataBind();
            }
        }

        public string SelectedDepartment
        {
            get { return this.DropDownListDepartments.SelectedValue; }
        }

        public IEnumerable<Employee> Employees
        {
            set
            {
                this.GridViewEmployees.DataSource = value;
                this.GridViewEmployees.DataBind();
            }
        }
    }

```

PV 模式将所有的 UI 处理逻辑全部定义在 **Presenter** 上，意味着所有的 UI 处理逻辑都可以被测试，所以从可测试性的角度来这是一种不错的选择，但是它要求将 **View** 中可供操作的 UI 元素定义在对应的接口中，对于一些复杂的富客户端（Rich Client）**View** 来说，接口成员将会变得很多，这无疑会提升编程所需的代码量。从另一方面来看，由于 **Presenter** 需要在控件级别对 **View** 进行细粒度的控制，这无疑会提供 **Presenter** 本身的复杂度，往往会使原本简单的逻辑复杂化，在这种情况下我们往往采用 SC 模式。

在 SC 模式下，为了降低 **Presenter** 的复杂度，我们将诸如数据绑定和格式化这样简单的 UI 处理逻辑转移到 **View** 中，这些处理逻辑会体现在 **View** 实现的接口中。尽管 **View** 从 **Presenter** 中接管了部分 UI 处理逻辑，但是 **Presenter** 依然是整个三角关系的驱动者，**View** 被动的地位依然没有改变。对于用户作用在 **View** 上的交互操作，**View** 本身并不进行响应，而是直接将交互请求转发给 **Presenter**，后者在独立完成相应的处理流程（可能涉及针对 **Model** 的调用）之后会驱动 **View** 或者创建新的 **View** 作为对用户交互操作的响应。

View 和 Presenter 交互的规则（针对 SC 模式）

View 和 **Presenter** 之间的交互是整个 MVP 的核心，能否正确地应用 MVP 模式来架构我们的应用主要取决于能否正确地处理 **View** 和 **Presenter** 两者之间的关系。在由 **Model**、**View**

和 Presenter 组成的三角关系中,核心不是 View 而是 Presenter,Presenter 不是 View 调用 Model 的中介,而是最终决定如何响应用户交互行为的决策者。

打个比方,View 是 Presenter 委派到前端的客户代理,而作为客户的自然就是最终的用户。对于以鼠标/键盘操作体现的交互请求应该如何处理,作为代理的 View 并没有决策权,所以它会将请求汇报给委托人 Presenter。View 向 Presenter 发送用户交互请求应该采用这样的口吻:“我现在将用户交互请求发送给你,你看着办,需要我的时候我会协助你”,而不应该是这样:“我现在处理用户交互请求了,我知道该怎么办,但是我需要你的支持,因为实现业务逻辑的 Model 只信任你”。

对于 Presenter 处理用户交互请求的流程,如果中间环节需要涉及到 Model,它会直接发起对 Model 的调用。如果需要 View 的参与(比如需要将 Model 最新的状态反应在 View 上),Presenter 会驱动 View 完成相应的工作。

对于绑定到 View 上的数据,不应该是 View 从 Presenter 上“拉”回来的,应该是 Presenter 主动“推”给 View 的。从消息流(或者消息交换模式)的角度来讲,不论是 View 向 Presenter 完成针对用户交互请求的通知,还是 Presenter 在进行交互请求处理过程中驱动 View 完成相应的 UI 操作,都是单向(One-Way)的。反应在应用编程接口的定义上就意味着不论是定义在 Presenter 中被 View 调用的方法,还是定义在 IView 接口中被 Presenter 调用的方法最好都没有返回值。如果不采用方法调用的形式,我们也可以通过事件注册的方式实现 View 和 Presenter 的交互,事件机制体现的消息流无疑是单向的。

View 本身仅仅实现单纯的、独立的 UI 处理逻辑,它处理的数据应该是 Presenter 实时推送给它的,所以 View 尽可能不维护数据状态。定义在 IView 的接口最好只包含方法,而避免属性的定义,Presenter 所需的关于 View 的状态应该在接收到 View 发送的用户交互请求的时候一次得到,而不需要通过 View 的属性去获取。

实例演示: SC 模式的应用(S101)

为了让读者对 MVP 模式,尤其是该模式下的 View 和 Presenter 之间的交互方式有一个深刻的认识,我们现在来做一个简单的实例演示。本实例采用上面提及的关于员工查询的场景,并且采用 ASP.NET Web Forms 来建立这个简单的应用,最终呈现出来的效果如图 1-3 所示。前面我们已经演示了采用 PV 模式下的 IView 应该如何定义,现在我们来看看 SC 模式下的 IView 有何不同。

先来看看表示员工信息的数据类型如何定义。我们通过具有如下定义的数据类型 Employee 来表示一个员工。简单起见,我们仅仅定义了表示员工基本信息(ID、姓名、性别、出生日期和部门)的 5 个属性。

```
public class Employee
{
    public string      Id { get; private set; }
```

```

public string      Name { get; private set; }
public string      Gender { get; private set; }
public DateTime    BirthDate { get; private set; }
public string      Department { get; private set; }

public Employee(string id, string name, string gender,
    DateTime birthDate, string department)
{
    this.Id          = id;
    this.Name        = name;
    this.Gender      = gender;
    this.BirthDate   = birthDate;
    this.Department  = department;
}
}

```

作为包含应用状态和状态操作行为的 Model 通过如下一个简单的 `EmployeeRepository` 类型来体现。如代码所示,表示所有员工列表的数据通过一个静态字段来维护,而 `GetEmployees` 返回指定部门的员工列表,如果没有指定筛选部门或者指定的部门字符为空,则直接返回所有的员工列表。

```

public class EmployeeRepository
{
    private static IList<Employee> employees;
    static EmployeeRepository()
    {
        employees = new List<Employee>();
        employees.Add(new Employee("001", "张三", "男",
            new DateTime(1981, 8, 24), "销售部"));
        employees.Add(new Employee("002", "李四", "女",
            new DateTime(1982, 7, 10), "人事部"));
        employees.Add(new Employee("003", "王五", "男",
            new DateTime(1981, 9, 21), "人事部"));
    }

    public IEnumerable<Employee> GetEmployees(string department = "")
    {
        if (string.IsNullOrEmpty(department))
        {
            return employees;
        }
        return employees.Where(e => e.Department == department).ToArray();
    }
}

```

接下来我们来看作为 View 接口的 `IEmployeeSearchView` 的定义。如下面的代码片段所示,该接口定义了 `BindEmployees` 和 `BindDepartments` 两个方法,分别用于绑定基于部门列表的 `DropDownList` 和基于员工列表的 `GridView`。除此之外, `IEmployeeSearchView` 接口还定义了一个事件 `DepartmentSelected`,该事件会在用户选择了筛选部门后点击“查询”按钮时触发。`DepartmentSelected` 事件参数类型为自定义的 `DepartmentSelectedEventArgs`,属性 `Department` 表示用户选择的部门。

```

public interface IEmployeeSearchView
{

```

```

void BindEmployees(IEnumerable<Employee> employees);
void BindDepartments(IEnumerable<string> departments);
event EventHandler<DepartmentSelectedEventArgs> DepartmentSelected;
}

public class DepartmentSelectedEventArgs : EventArgs
{
    public string Department { get; private set; }
    public DepartmentSelectedEventArgs(string department)
    {
        this.Department = department;
    }
}

```

作为 MVP 三角关系核心的 Presenter 通过 EmployeeSearchPresenter 表示。如下面的代码片段所示，表示 View 的只读属性类型为 IEmployeeSearchView 接口，而另一个只读属性 Repository 则表示作为 Model 的 EmployeeRepository 对象，两个属性均在构造函数中初始化。

```

public class EmployeeSearchPresenter
{
    public IEmployeeSearchView View { get; private set; }
    public EmployeeRepository Repository { get; private set; }

    public EmployeeSearchPresenter(IEmployeeSearchView view)
    {
        this.View = view;
        this.Repository = new EmployeeRepository();
        this.View.DepartmentSelected += OnDepartmentSelected;
    }

    public void Initialize()
    {
        IEnumerable<Employee> employees = this.Repository.GetEmployees();
        this.View.BindEmployees(employees);
        string[] departments =
            new string[] { "销售部", "采购部", "人事部", "IT 部" };
        this.View.BindDepartments(departments);
    }

    protected void OnDepartmentSelected(object sender,
        DepartmentSelectedEventArgs args)
    {
        string department = args.Department;
        var employees = this.Repository.GetEmployees(department);
        this.View.BindEmployees(employees);
    }
}

```

在构造函数中我们注册了 View 的 DepartmentSelected 事件，作为事件处理器的 OnDepartmentSelected 方法通过调用 Repository（即 Model）得到了用户选择部门下的员工列表，返回的员工列表通过调用 View 的 BindEmployees 方法实现了在 View 上的数据绑定。在 Initialize 方法中，我们通过调用 Repository 获取所有员工的列表，并通过 View 的 BindEmployees 方法显示在界面上。作为筛选条件的部门列表通过调用 View 的 BindDepartments 方法绑定在 View 上。

最后我们来看看作为 View 的 Web 页面如何定义。如下所示的是作为页面主体部分的 HTML，核心部分是一个用于绑定筛选部门列表的 DropDownList 和一个绑定员工列表的 GridView。

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>员工管理</title>
    <link rel="stylesheet" href="Style.css" />
  </head>
  <body>
    <form id="form1" runat="server">
      <div id="page">
        <div class="top">
          选择查询部门:
          <asp:DropDownList ID="DropDownListDepartments"
            runat="server" />
          <asp:Button ID="ButtonSearch" runat="server" Text="查询"
            OnClick="ButtonSearch_Click" />
        </div>
        <asp:GridView ID="GridViewEmployees" runat="server"
          AutoGenerateColumns="false" Width="100%">
          <Columns>
            <asp:BoundField DataField="Name" HeaderText="姓名" />
            <asp:BoundField DataField="Gender" HeaderText="性别" />
            <asp:BoundField DataField="BirthDate"
              HeaderText="出生日期"
              DataFormatString="{0:dd/MM/yyyy}" />
            <asp:BoundField DataField="Department" HeaderText="部门"/>
          </Columns>
        </asp:GridView>
      </div>
    </form>
  </body>
</html>
```

如下所示的是该 Web 页面的后台代码的定义，它实现了定义在 IEmployeeSearchView 接口的两个方法（BindEmployees 和 BindDepartments）和一个事件（DepartmentSelected）。表示 Presenter 的同名只读属性在构造函数中被初始化。在页面加载的时候（Page_Load 方法）Presenter 的 Initialize 方法被调用，而在“查询”按钮被点击的时候（ButtonSearch_Click）事件 DepartmentSelected 被触发。

```
public partial class Default : Page, IEmployeeSearchView
{
    public EmployeeSearchPresenter Presenter { get; private set; }
    public event EventHandler<DepartmentSelectedEventArgs> DepartmentSelected;

    public Default()
    {
        this.Presenter = new EmployeeSearchPresenter(this);
    }

    protected void Page_Load(object sender, EventArgs e)
    {
```

```

        if (!this.IsPostBack)
        {
            this.Presenter.Initialize();
        }
    }

    protected void ButtonSearch_Click(object sender, EventArgs e)
    {
        string department = this.DropDownListDepartments.SelectedValue;
        DepartmentSelectedEventArgs eventArgs =
            new DepartmentSelectedEventArgs(department);
        if (null != DepartmentSelected)
        {
            DepartmentSelected(this, eventArgs);
        }
    }

    public void BindEmployees(IEnumerable<Employee> employees)
    {
        this.GridViewEmployees.DataSource = employees;
        this.GridViewEmployees.DataBind();
    }

    public void BindDepartments(IEnumerable<string> departments)
    {
        this.DropDownListDepartments.DataSource = departments;
        this.DropDownListDepartments.DataBind();
    }
}

```

1.2.2 Model 2

Trygve M. H. Reenskau 当初提出的 MVC 是作为基于 GUI 的桌面应用的架构模式并不太适合 Web 本身的特性，虽然 MVC/MVP 也可以直接用于 ASP.NET Web Forms 应用，但这是因为微软就是基于桌面应用的编程模式来设计基于 Web Forms 的 ASP.NET 应用框架的。Web 应用不同于 GUI 桌面应用的主要区别在于：用户是通过浏览器与应用进行交互，交互请求和响应是通过 HTTP 请求和响应来完成的。

为了让 MVC 能够为 Web 应用提供原生的支持，另一个被称为 Model 2 的 MVC 变体被提出来，这来源于基于 Java 的 Web 应用架构模式。Java Web 应用具有两种基本的基于 MVC 的架构模式，分别被称为 Model 1 和 Model 2。Model 1 类似于我们前面提及的自测试图模式，它将数据的可视化呈现和用户交互操作的处理逻辑合并在一起。Model 1 使用于那些比较简单的 Web 应用，对于相对复杂的应用应该采用 Model 2。

为了让开发者采用相同的编程模式进行 GUI 桌面应用和 Web 应用的开发，微软通过 ViewState 和 Postback 对 HTTP 请求和回复机制进行了封装，使我们能够像编写 Windows Forms 应用一样采用事件驱动的方式进行 ASP.NET Web Forms 应用的编程。而 Model 2 采用完全不同的设计，它让开发者直接面向 Web，让他们关注 HTTP 的请求和响应，所以 Model 2 提供对 Web 应用原生的支持。

对于 Web 应用来说, 和用户直接交互的 UI 界面由浏览器来提供, 用户交互请求通过浏览器以 HTTP 请求的方式发送到 Web 服务器, 服务器对请求进行相应的处理并最终返回一个 HTTP 回复对请求予以响应。接下来我们详细讨论作为 MVC 的三要素是如何相互协作最终完成对请求的响应的。图 1-4 所示的序列图体现了整个流程的全过程。

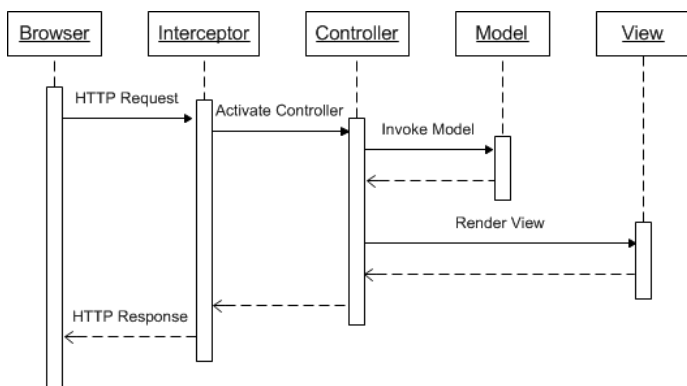


图 1-4 Model 2 交互流程

Model 2 中一个 HTTP 请求的目标是 Controller 中的某个 Action, 后者体现为定义在 Controller 类型中的某个方法, 所以对请求的处理最终体现在对目标 Controller 对象的激活和对相应 Action 方法的执行。一般来说, Controller 的类型和 Action 方法的名称以及作为 Action 方法的部分参数 (针对 HTTP-GET) 可以直接通过请求的 URL 解析出来。

如图 1-4 所示, 我们通过一个拦截器 (Interceptor) 对抵达 Web 服务器的 HTTP 请求进行拦截。一般的 Web 应用框架都提供了这样的拦截机制, 对于 ASP.NET 来说, 我们可以通过 `HttpModule` 的形式来定义这么一个拦截器。拦截器根据请求解析出目标 Controller 的类型和对应的 Action 方法的名称, 随后目标 Controller 被激活, 相应的 Action 方法被执行。

在激活 Controller 对象的目标 Action 方法被执行过程中, 它可以调用 Model 获取相应的数据或者改变其状态。在 Action 方法执行的最后阶段会选择相应的 View, 整个 View 被最终转换成 HTML, 以 HTTP 响应的形式返回到客户端并呈现在浏览器中。绑定在 View 上的数据来源于 Model 或者基于显示要求进行的简单逻辑计算, 我们有时候将它们称为 VM (View Model), 即基于 View 的 Model (这里的 View Model 与 MVVM 模式下的 VM 是完全不同的两个概念, 后者不仅包括呈现在 View 中的数据, 也包括数据操作行为)。

1.2.3 ASP.NET MVC 与 Model 2

ASP.NET MVC 就是根据 Model 2 模式设计的。对于 HTTP 请求的拦截以实现为目标 Controller 和 Action 的解析是通过一个自定义 `HttpModule` 来实现的, 而对目标 Controller 的激活则通过一个自定义 `Handler` 来完成。在本章的最后我们会通过一个例子来模拟 ASP.NET MVC 的工作原理。

在上面我们多次强调 MVC 的 Model 是维持应用状态提供业务功能的领域模型，或者是多层架构中进入业务层的入口或者业务服务的代理，但是 ASP.NET MVC 中的 Model 还是这个 Model 吗？稍微了解 ASP.NET MVC 的读者都知道，ASP.NET MVC 的 Model 仅仅是绑定到 View 上的数据而已，它和 MVC 模式中的 Model 并不是一回事。由于 ASP.NET MVC 中的 Model 是基于 View 的，我们可以将其称为 View Model。

由于 ASP.NET MVC 只有 View Model，所以 ASP.NET MVC 应用框架本身仅仅关于 View 和 Controller，真正的 Model 以及 Model 和 Controller 之间的交互体现在我们如何来设计 Controller。我个人觉得将用于构建 ASP.NET MVC 的 MVC 模式成为 M (Model) -V (View) -VM (View Model) -C (Controller) 也许更为准确。

1.3 IIS/ASP.NET 管道

前面我们对 MVC 模式及其变体作了详细的介绍，其目的在于让读者充分地了解 ASP.NET MVC 框架的设计思想，接下来我们来介绍支撑 ASP.NET MVC 的技术平台。顾名思义，ASP.NET MVC 就是建立在 ASP.NET 平台上基于 MVC 模式建立的 Web 应用框架，深刻理解 ASP.NET MVC 的前提是对 ASP.NET 管道式设计具有深刻的认识。由于 ASP.NET Web 应用总是寄宿于 IIS 上，所以我们将两者结合起来介绍，力求让读者完整地理解请求在 IIS/ASP.NET 管道中是如何流动的。由于不同版本的 IIS 的处理方式具有很大的差异，接下来会介绍 3 个主要的 IIS 版本各自对 Web 请求的不同处理方式。

1.3.1 IIS 5.x 与 ASP.NET

我们先来看看 IIS 5.x 是如何处理基于 ASP.NET 资源（比如.aspx、.asmx 等）请求的。整个过程基本上可以通过图 1-5 体现。IIS 5.x 运行在进程 InetInfo.exe 中，该进程寄宿着一个名为 World Wide Web Publishing Service（简称 W3SVC）的 Windows 服务。W3SVC 的主要功能包括 HTTP 请求的监听、工作进程和配置管理（通过从 Metabase 中加载相关配置信息）等。

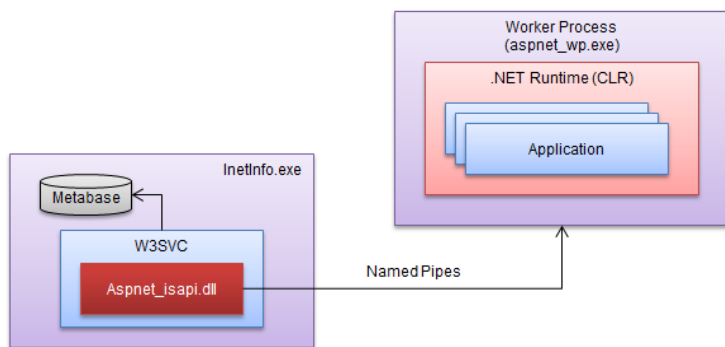


图 1-5 IIS 5.x 与 ASP.NET

当检测到某个 HTTP 请求时，先根据扩展名判断请求的是否是静态资源（比如.html、.img、.txt、.xml等），如果是，则直接将文件内容以 HTTP 回复的形式返回；如果是动态资源（比如.aspx、.asp、.php等），则通过扩展名从 IIS 的脚本映射（Script Map）中找到相应的 ISAPI 动态连接库（Dynamic Link Library, DLL）。

ISAPI（Internet Server Application Programming Interface）是一套本地的（Native）Win32 API，是 IIS 和其他动态 Web 应用或平台之间的纽带。ISAPI 定义在一个动态连接库（DLL）文件中，ASP.NET ISAPI 对应的 DLL 文件名称为 `aspnet_isapi.dll`，我们可以在目录“`%windir%\Microsoft.NET\Framework\{version no}\`”中找到它。ISAPI 支持 ISAPI 扩展（ISAPI Extension）和 ISAPI 筛选（ISAPI Filter），前者是真正处理 HTTP 请求的接口，后者则可以在 HTTP 请求真正被处理之前查看、修改、转发或拒绝请求，比如 IIS 可以利用 ISAPI 筛选进行请求的验证。

如果我们请求的是一个基于 ASP.NET 的资源类型，比如.aspx、.asmx 和.svc 等，`aspnet_isapi.dll` 会被加载，而 ASP.NET ISAPI 扩展会创建 ASP.NET 的工作进程（如果该进程尚未启动）。对于 IIS 5.x 来说，该工作进程为 `aspnet.exe`。IIS 进程与工作进程之间通过命名管道（Named Pipes）进行通信。

在工作进程初始化过程中，.NET 运行时（CLR）被加载进而构建了一个托管的环境。对于某个 Web 应用的初次请求，CLR 会为其创建一个应用程序域（Application Domain）。在应用程序域中，HTTP 运行时（HTTP Runtime）被加载并用以创建相应的应用。寄宿于 IIS 5.x 的所有 Web 应用都运行在同一个进程（工作进程 `aspnet_wp.exe`）的不同应用程序域中。

1.3.2 IIS 6.0 与 ASP.NET

通过上面的介绍，我们可以看出 IIS 5.x 至少存在着如下两个方面的不足。

- ISAPI 动态连接库被加载到 `InetInfo.exe` 进程中，它和工作进程之间是一种典型的跨进程通信方式，尽管采用命名管道，但是仍然会带来性能的瓶颈。
- 所有的 ASP.NET 应用运行在相同进程（`aspnet_wp.exe`）中的不同的应用程序域中，基于应用程序域的隔离不能从根本上解决一个应用程序对另一个程序的影响。在更多的时候，我们需要不同的 Web 应用运行在不同的进程中。

为了解决第一个问题，IIS 6.0 将 ISAPI 动态连接库直接加载到工作进程中；为了解决第二个问题，引入了应用程序池（Application Pool）的机制。我们可以为一个或多个 Web 应用创建应用程序池，由于每一个应用程序池对应一个独立的工作进程，从而为运行在不同应用程序池中的 Web 应用提供基于进程的隔离级别。IIS 6.0 的工作进程名称为 `w3wp.exe`。

除了上面两点改进之外，IIS 6.0 还有其他一些值得称道的地方。其中最重要的一点就是创建了一个名为 HTTP.SYS 的 HTTP 监听器。HTTP.SYS 以驱动程序的形式运行在 Windows 的内核模式（Kernel Mode）下，它是 Windows 2003 的 TCP/IP 网络子系统的一部分，从结构上看它属于 TCP 之上的一个网络驱动程序。

严格地说，HTTP.SYS 已经不属于 IIS 的范畴了，所以 HTTP.SYS 的配置信息也没有保存在 IIS 的元数据库（Metabase）中，而是定义在注册表中。HTTP.SYS 的注册表项的路径为 HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/HTTP。HTTP.SYS 能够带来如下的好处。

- 持续监听：由于 HTTP.SYS 是一个网络驱动程序，始终处于运行状态，对于用户的 HTTP 请求能够及时作出反应。
- 更好的稳定性：HTTP.SYS 运行在操作系统内核模式下，并不执行任何用户代码，所以其本身不会受到 Web 应用、工作进程和 IIS 进程的影响。
- 内核模式下数据缓存：如果某个资源被频繁请求，HTTP.SYS 会把响应的内容进行缓存，缓存的内容可以直接响应后续的请求。由于这是基于内核模式的缓存，不存在内核模式 and 用户模式的切换，响应速度将得到极大的改进。

图 1-6 体现了 IIS 的结构和处理 HTTP 请求的流程。与 IIS 5.x 不同，W3SVC 从 InetInfo.exe 进程脱离出来（对于 IIS 6.0 来说，InetInfo.exe 基本上可以看作单纯的 IIS 管理进程），运行在另一个进程 SvcHost.exe 中。不过 W3SVC 的基本功能并没有发生变化，只是在功能的实现上作了相应的改进。与 IIS 5.x 一样，元数据库（Metabase）依然存在于 InetInfo.exe 进程中。

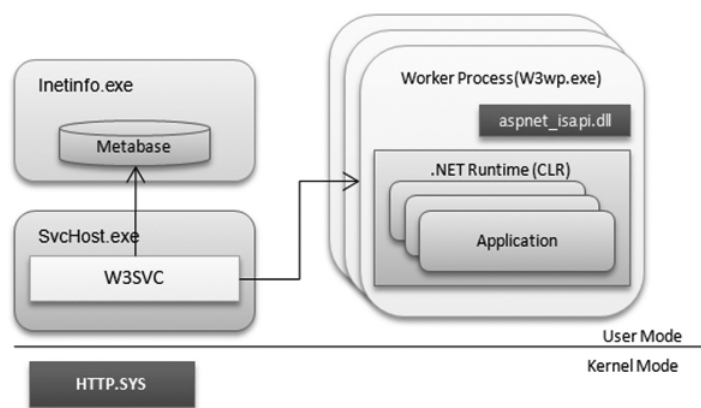


图 1-6 IIS 6.0 与 ASP.NET

当 HTTP.SYS 监听到用户的 HTTP 请求时将其分发给 W3SVC，W3SVC 解析出请求的 URL，并根据从 Metabase 获取的 URL 与 Web 应用之间的映射关系得到目标应用，并进一步

得到目标应用运行的应用程序池或工作进程。如果工作进程不存在（尚未创建或被回收），则为该请求创建新的工作进程。我们将工作进程的这种创建方式称为请求式创建。在工作进程的初始化过程中，相应的 ISAPI 动态连接库被加载。对于 ASP.NET 应用来说，被加载的 ISAPI.dll 为 `aspnet_isapi.dll`。ASP.NET ISAPI 再负责进行 CLR 的加载、应用程序域的创建和 Web 应用的初始化等操作。

1.3.3 IIS 7.0 与 ASP.NET

IIS 7.0 在请求的监听和分发机制上又进行了革新性的改进，主要体现在对于 Windows 进程激活服务（Windows Process Activation Service, WAS）的引入，将原来（IIS 6.0）W3SVC 承载的部分功能分流给了 WAS。通过上面的介绍，我们知道对于 IIS 6.0 来说 W3SVC 主要承载着 3 大功能。

- HTTP 请求接收：接收 HTTP.SYS 监听到的 HTTP 请求。
- 配置管理：从元数据库（Metabase）中加载配置信息对相关组件进行配置。
- 进程管理：创建、回收、监控工作进程。

IIS 7.0 将后两组功能实现到了 WAS 中，接收 HTTP 请求的任务依然落在 W3SVC 头上。WAS 的引入为 IIS 7.0 提供了对非 HTTP 协议的支持。WAS 通过监听器适配器接口（Listener Adapter Interface）抽象出不同协议监听器。具体来说，除了基于网络驱动的 HTTP.SYS 提供 HTTP 请求监听功能外还提供了 TCP 监听器、命名管道监听器和 MSMQ 监听器以提供基于 TCP、命名管道和 MSMQ 传输协议的监听支持。

与此 3 种监听器相对的是 3 种监听适配器，它们提供监听器与 WAS 中的监听器适配器接口之间的适配。从这个意义上讲，IIS 7.0 中的 W3SVC 更多地为 HTTP.SYS 起着监听适配器的作用。这 3 种非 HTTP 监听器和监听适配器定义在程序集 `SMHost.exe` 中，我们可以在目录 `%windir%\Microsoft.NET\Framework\v3.0\Windows Communication Foundation\` 中找到它们。

WCF 提供的这 3 种监听器和监听适配器最终以 Windows 服务的形式体现。虽然它们定义在一个程序集中，我们依然可以通过服务工作管理器对其进行单独的启动、终止和配置。`SMHost.exe` 提供了 4 个重要的 Windows Service。

- `NetTcpPortSharing`：为 WCF 提供 TCP 端口共享。关于端口共享在 WCF 中的应用，本人拙著《WCF 全面解析》（上册）对此有详细的介绍。
- `NetTcpActivator`：为 WAS 提供基于 TCP 的激活请求，包含 TCP 监听器和对应的监听适配器。
- `NetPipeActivator`：为 WAS 提供基于命名管道的激活请求，包含命名管道监听器和对应的监听适配器。

- NetMsmqActivator: 为 WAS 提供基于 MSMQ 的激活请求, 包含 MSMQ 监听器和对应的监听适配器。

图 1-7 为上述的 4 个 Windows 服务在服务控制管理器中的呈现。

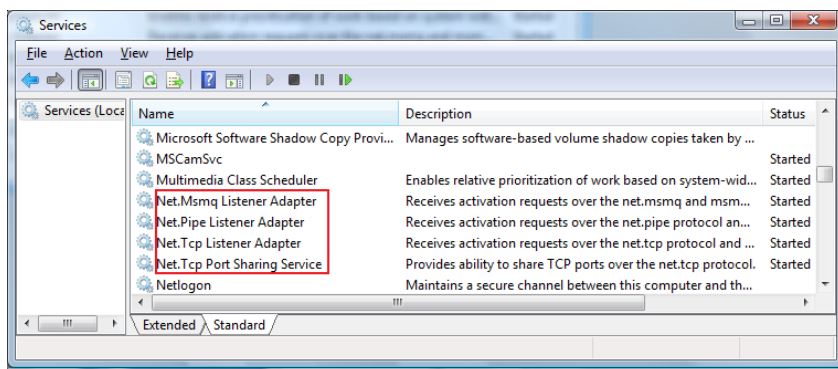


图 1-7 定义在 SMHost.exe 中的 Windows Service

图 1-8 揭示了 IIS 7.0 的整体构架及整个请求处理流程。无论是从 W3SVC 接收到的 HTTP 请求, 还是通过 WCF 提供的监听适配器接收到的请求, 最终都会传递到 WAS。如果相应的工作进程 (或者应用程序池) 尚未创建, 则创建它, 否则将请求分发给对应的工作进程进行后续的处理。WAS 在进行请求处理过程中, 通过内置的配置管理模块加载相关的配置信息, 并对相关的组件进行配置。与 IIS 5.x 和 IIS 6.0 基于 Metabase 的配置信息存储不同的是, IIS 7.0 大都将配置信息存放于 XML 形式的配置文件中, 基本的配置存放在 applicationHost.config 中。

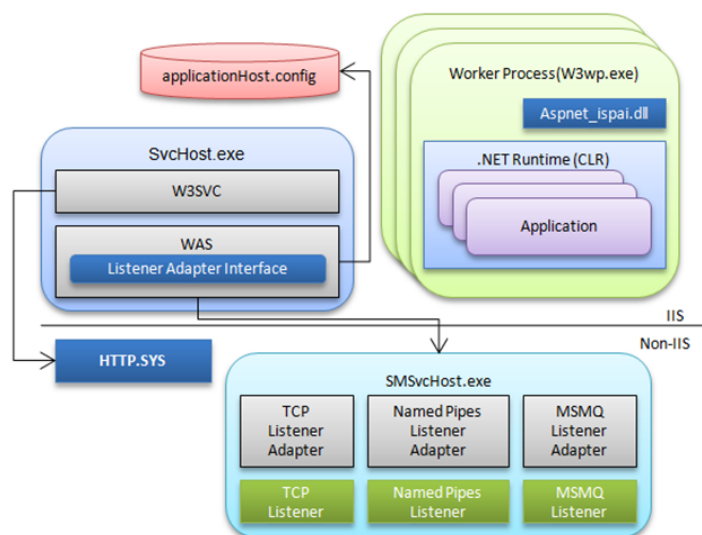


图 1-8 IIS 7.0 与 ASP.NET

ASP.NET 集成

从上面对 IIS 5.x 和 IIS 6.0 的介绍中，我们不难发现 IIS 与 ASP.NET 是两个相互独立的管道（Pipeline）。在各自管辖范围内，它们各自具有自己的一套机制对 HTTP 请求进行处理。两个管道通过 ISAPI 实现“连通”，IIS 是第一道屏障，当对 HTTP 请求进行必要的前期处理（比如身份验证等）时，通过 ISAPI 将请求分发给 ASP.NET 管道。当 ASP.NET 在自身管道范围内完成对 HTTP 请求的处理时，处理后的结果再返回到 IIS，IIS 对其进行后期处理（比如日志记录、压缩等），最终生成 HTTP 响应。图 1-9 反映了 IIS 6.0 与 ASP.NET 之间的桥接关系。

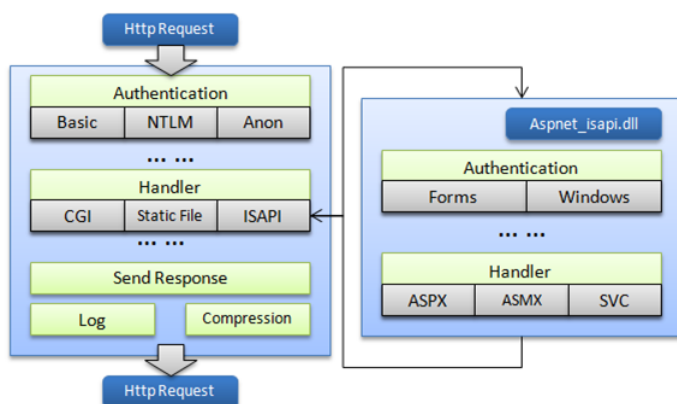


图 1-9 基于 IIS 6.0 与 ASP.NET 双管道设计

从另一个角度讲，IIS 运行在非托管的环境中，而 ASP.NET 管道则是托管的，ISAPI 还是连接非托管环境和托管环境的纽带。IIS 5.x 和 IIS 6.0 把两个管道进行隔离至少带来了下面的一些局限与不足：

- 相同操作的重复执行：IIS 与 ASP.NET 之间具有一些重复的操作，比如身份验证。
- 动态文件与静态文件处理的不一致：因为只有基于 ASP.NET 动态文件（比如.aspx、.asmx、.svc 等）的 HTTP 请求才能通过 ASP.NET ISAPI 进入 ASP.NET 管道，而对于一些静态文件（比如.html、.xml、.img 等）的请求则由 IIS 直接响应，那么 ASP.NET 管道中的一些功能将不能用于这些基于静态文件的请求，比如我们希望通过 Forms 认证应用于基于图片文件的请求就做不到。
- IIS 难以扩展：对于 IIS 的扩展基本上就体现在自定义 ISAPI，但是对于大部分人来说，这不是一件容易的事情。因为 ISAPI 是基于 Win32 的非托管的 API，并非一种面向应用的编程接口。通常我们希望的是诸如定义 ASP.NET 的 HttpModule 和 HttpHandler 一样，通过托管代码的方式来扩展 IIS。

对于 Windows 平台下的 IIS 来讲，ASP.NET 无疑是一等公民，它们之间不应该是“井水不犯河水”，而应该是“你中有我，我中有你”的关系，为此在 IIS 7.0 中实现了两者的集成，

通过集成可以获得如下的好处。

- 允许通过本地代码(Native Code)和托管代码(Managed Code)两种方式定义 IIS Module, 这些 IIS Module 注册到 IIS 中形成一个通用的请求处理管道。由这些 IIS Module 组成的这个管道能够处理所有的请求, 不论请求基于怎样的资源类型。比如, 可以将 FormsAuthenticationModule 提供的 Forms 认证应用到基于 .aspx、CGI 和静态文件的请求。
- 将 ASP.NET 提供的一些强大的功能应用到原来难以企及的地方, 比如将 ASP.NET 的 URL 重写功能置于身份验证之前。
- 采用相同的方式去实现、配置、检测和支持一些服务器特性 (Feature), 比如 Module、Handler 映射、定制错误配置 (Custom Error Configuration) 等。

图 1-10 演示了在 ASP.NET 集成模式下, IIS 整个请求处理管道的结构。可以看到, 原来 ASP.NET 提供的托管组件可以直接应用在 IIS 管道中。

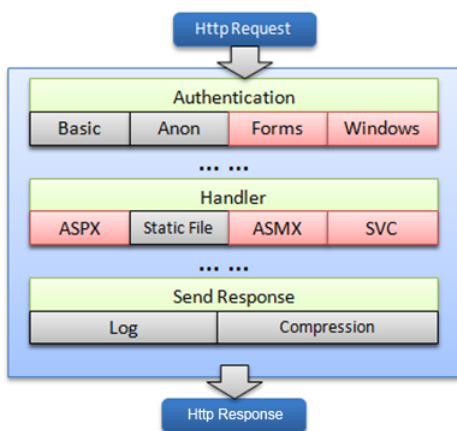


图 1-10 基于 IIS 7.0 与 ASP.NET 集成管道设计

1.3.4 ASP.NET 管道

以 IIS 6.0 为例, 在工作进程 w3wp.exe 中, 利用 aspnet_isapi.dll 加载 .NET 运行时 (如果 .NET 运行时尚未加载), IIS 6.0 引入了应用程序池的概念, 一个工作进程对应着一个应用程序池。一个应用程序池可以承载一个或多个 Web 应用, 每个 Web 应用映射到一个 IIS 虚拟目录。与 IIS 5.x 一样, 每一个 Web 应用运行在各自的应用程序域中。

如果 HTTP.SYS 接收到的 HTTP 请求是对该 Web 应用的第一次访问, 在成功加载了运行时后, 会通过 AppDomainFactory 为该 Web 应用创建一个应用程序域, 随后一个特殊的运行时 IsapiRuntime 被加载。IsapiRuntime 定义在程序集 System.Web 中, 对应的命名空间为 System.Web.Hosting, 被加载的 IsapiRuntime 会接管该 HTTP 请求。

IsapiRuntime 会首先创建一个 IsapiWorkerRequest 对象，用于封装当前的 HTTP 请求，并将该 IsapiWorkerRequest 对象传递给 ASP.NET 运行时 HttpRuntime。从此时起，HTTP 请求正式进入了 ASP.NET 管道。HttpRuntime 会根据 IsapiWorkerRequest 对象创建用于表示当前 HTTP 请求的上下文（Context）对象 HttpContext。

随着 HttpContext 被成功创建，HttpRuntime 会利用 HttpApplicationFactory 创建新的或获取现有的 HttpApplication 对象。实际上 ASP.NET 维护着一个 HttpApplication 对象池，HttpApplicationFactory 从池中选取可用的 HttpApplication 用于处理 HTTP 请求，处理完后将其释放到对象池中。HttpApplicationFactory 负责处理当前的 HTTP 请求。

在 HttpApplication 初始化过程中，会根据配置文件加载并初始化相应的 HttpModule 对象。对于 HttpApplication 来说，在它处理 HTTP 请求的不同阶段会触发不同的事件（Event），而 HttpModule 的意义在于通过注册 HttpApplication 的相应的事件，将所需的操作注入整个 HTTP 请求的处理流程。ASP.NET 的很多功能，比如身份验证、授权、缓存等，都是通过相应的 HttpModule 实现的。

最终完成对 HTTP 请求的处理实现在 HttpHandler 中。对于不同的资源类型，具有不同的 HttpHandler。比如.aspx 页面对应的 HttpHandler 为 System.Web.UI.Page，WCF 的.svc 文件对应的 HttpHandler 为 System.ServiceModel.Activation.HttpHandler。上面整个处理流程如图 1-11 所示。

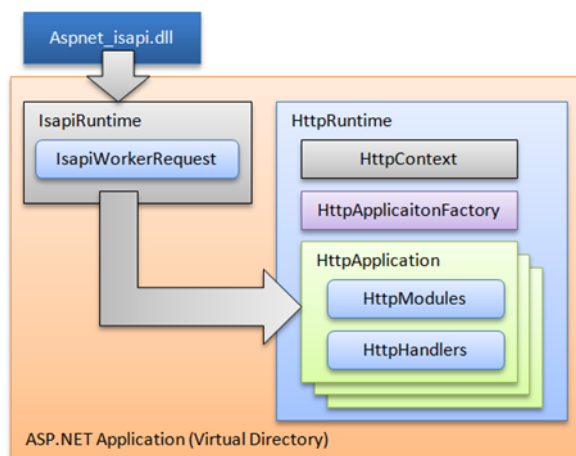


图 1-11 ASP.NET 处理管道

HttpApplication

HttpApplication 是整个 ASP.NET 基础架构的核心，它负责处理分发给它的 HTTP 请求。由于一个 HttpApplication 对象在某个时刻只能处理一个请求，只有完成对某个请求的处理后，HttpApplication 才能用于后续的请求的处理，所以 ASP.NET 采用对象池的机制来创建或

获取 `HttpApplication` 对象。

当第一个请求抵达时，`ASP.NET` 会一次创建多个 `HttpApplication` 对象，并将其置于池中，选择其中一个对象来处理该请求。处理完毕后，`HttpApplication` 不会被回收，而是释放到池中。对于后续的请求，空闲的 `HttpApplication` 对象会从池中取出，如果池中所有的 `HttpApplication` 对象都处于繁忙的状态，`ASP.NET` 会创建新的 `HttpApplication` 对象。

`HttpApplication` 处理请求的整个生命周期是一个相对复杂的过程，在该过程的不同阶段会触发相应的事件。我们可以注册相应的事件，将处理逻辑注入到 `HttpApplication` 处理请求的某个阶段。表 1-1 按照实现的先后顺序列出了 `HttpApplication` 在处理每一个请求时触发的事件名称。

表 1-1 `HttpApplication` 事件列表

名 称	描 述
<code>BeginRequest</code>	<code>HTTP</code> 管道开始处理请求时，会触发 <code>BeginRequest</code> 事件
<code>AuthenticateRequest</code> , <code>PostAuthenticateRequest</code>	<code>ASP.NET</code> 先后触发这两个事件，使安全模块对请求进行身份验证
<code>AuthorizeRequest</code> , <code>PostAuthorizeRequest</code>	<code>ASP.NET</code> 先后触发这两个事件，使安全模块对请求进程授权
<code>ResolveRequestCache</code> , <code>PostResolveRequestCache</code>	<code>ASP.NET</code> 先后触发这两个事件，以使缓存模块利用缓存的内容对请求直接进行响应（缓存模块可以将响应内容进行缓存，对于后续的请求，直接将缓存的内容返回，从而提高响应能力）
<code>PostMapRequestHandler</code>	对于访问不同的资源类型， <code>ASP.NET</code> 具有不同的 <code>HttpHandler</code> 对其进行处理。对于每个请求， <code>ASP.NET</code> 会通过扩展名选择匹配相应的 <code>HttpHandler</code> 类型，成功匹配后，该实现被触发
<code>AcquireRequestState</code> , <code>PostAcquireRequestState</code>	<code>ASP.NET</code> 先后触发这两个事件，使状态管理模块获取基于当前请求相应的状态，如 <code>SessionState</code>
<code>PreRequestHandlerExecute</code> , <code>PostRequestHandlerExecute</code>	<code>ASP.NET</code> 最终通过与请求资源类型相对应的 <code>HttpHandler</code> 实现对请求的处理，在实行 <code>HttpHandler</code> 前后，这两个实现被先后触发
<code>ReleaseRequestState</code> , <code>PostReleaseRequestState</code>	<code>ASP.NET</code> 先后触发这两个事件，使状态管理模块释放基于当前请求相应的状态
<code>UpdateRequestCache</code> , <code>PostUpdateRequestCache</code>	<code>ASP.NET</code> 先后触发这两个事件，以使缓存模块将 <code>HttpHandler</code> 处理请求得到的内容得以保存到输出缓存中
<code>LogRequest</code> , <code>PostLogRequest</code>	<code>ASP.NET</code> 先后触发这两个事件为当前请求进行日志记录
<code>EndRequest</code>	整个请求处理完成后， <code>EndRequest</code> 事件被触发

对于一个 `ASP.NET` 应用来说，`HttpApplication` 派生于 `Global.asax` 文件，我们可以通过创建 `global.asax` 文件对 `HttpApplication` 的请求处理行为进行定制。`Global.asax` 采用一种很直

接的方式实现了这样的功能，这种方式不是我们常用的方法重写或事件注册，而是直接采用方法名匹配。在 Global.asax 中，我们按照 “Application_{Event Name}” 这样的方法命名规则进行事件注册。比如 Application_BeginRequest 方法用于处理 HttpApplication 的 BeginRequest 事件。如果通过 VS 创建一个 Global.asax 文件，将采用如下的默认定义。

```
<%@ Application Language="C#" %>
<script runat="server">
    void Application_Start(object sender, EventArgs e){}
    void Application_End(object sender, EventArgs e){}
    void Application_Error(object sender, EventArgs e){}
    void Session_Start(object sender, EventArgs e){}
    void Session_End(object sender, EventArgs e){}
</script>
```

HttpModule

ASP.NET 拥有一个具有高度可扩展性的引擎，并且能够处理对于不同资源类型的请求，那么是什么成就了 ASP.NET 的高可扩展性呢？HttpModule 功不可没。

当请求转入 ASP.NET 管道时，最终负责处理该请求的是与请求资源类型相匹配的 HttpHandler 对象，但是在 Handler 正式工作之前，ASP.NET 会先加载并初始化所有配置的 HttpModule 对象。HttpModule 在初始化的过程中，会将一些功能注册到 HttpApplication 相应的事件中，在 HttpApplication 请求处理生命周期中的某个阶段，相应的事件会被触发，通过 HttpModule 注册的事件处理程序也得以执行。

所有的 HttpModule 都实现了具有如下定义的 System.Web.IHttpModule 接口，其中 Init 方法用于实现 HttpModule 自身的初始化，该方法接受一个 HttpApplication 对象，有了这个对象，事件注册就很容易了。

```
public interface IHttpModule
{
    void Dispose();
    void Init(HttpApplication context);
}
```

ASP.NET 提供的很多基础功能都是通过相应的 HttpModule 实现的，下面列出了一些典型的 HttpModule。

- OutputCacheModule: 实现了输出缓存（Output Caching）的功能。
- SessionStateModule: 在无状态的 HTTP 协议上实现了基于会话（Session）的状态。
- WindowsAuthenticationModule+FormsAuthenticationModule+PassportAuthenticationModule: 实现了 Windows、Forms 和 Passport 这 3 种典型的身份认证方式。
- UrlAuthorizationModule + FileAuthorizationModule: 实现了基于 URI 和文件 ACL（Access Control List）的授权。

除了这些系统定义的 `HttpModule` 之外,我们还可以自定义 `HttpModule`,通过 `Web.config` 可以很容易地将其注册到 Web 应用中。

HttpHandler

对于不同资源类型的请求,ASP.NET 会加载不同的 `Handler` 来处理,也就是说.aspx 页面与.aspx web 服务对应的 `Handler` 是不同的。所有的 `HttpHandler` 都实现了具有如下定义的接口 `System.Web.IHttpHandler`,方法 `ProcessRequest` 提供了处理请求的实现。

```
public interface IHttpHandler
{
    void ProcessRequest(HttpContext context);
    bool IsReusable { get; }
}
```

某些 `HttpHandler` 具有一个与之相关的 `HttpHandlerFactory`,它实现了具有如下定义的接口 `System.Web.IHttpHandlerFactory`,方法 `GetHandler` 用于创建新的 `HttpHandler`,或者获取已经存在的 `HttpHandler`。

```
public interface IHttpHandlerFactory
{
    IHttpHandler GetHandler(HttpContext context, string requestType,
        string url, string pathTranslated);
    void ReleaseHandler(IHttpHandler handler);
}
```

`HttpHandler` 和 `HttpHandlerFactory` 的类型都可以通过相同的方式配置到 `Web.config` 中。下面一段配置包含对.aspx、.asmx 和.svc 这 3 种典型的资源类型的 `HttpHandler` 配置。

```
<configuration>
  <system.web>
    <httpHandlers>
      <add path="*.svc"
        verb="*"
        type="System.ServiceModel.Activation.HttpHandler,
          System.ServiceModel, Version=4.0.0.0, Culture=neutral,
          PublicKeyToken=b77a5c561934e089"
        validate="false"/>
      <add path="*.aspx"
        verb="*"
        type="System.Web.UI.PageHandlerFactory"
        validate="true"/>
      <add path="*.asmx"
        verb="*"
        type="System.Web.Services.Protocols.WebServiceHandlerFactory,
          System.Web.Services, Version=4.0.0.0, Culture=neutral,
          PublicKeyToken=b03f5f7f11d50a3a"
        validate="False"/>
    </httpHandlers>
  </system.web>
</configuration>
```

1.4 ASP.NET MVC 是如何运行的

ASP.NET 由于采用了管道式设计，具有很好的扩展性，而整个 ASP.NET MVC 应用框架就是通过扩展 ASP.NET 实现的。通过上面对 ASP.NET 管道设计的介绍我们知道，ASP.NET 的扩展点主要体现在 `HttpModule` 和 `HttpHandler` 这两个核心组件之上，实际上整个 ASP.NET MVC 框架就是通过自定义的 `HttpModule` 和 `HttpHandler` 建立起来的。

为了使读者能够从整体上把握 ASP.NET MVC 的工作机制，接下来按照其原理通过一些自定义组件来模拟 ASP.NET MVC 的运行原理，也可以将此视为一个“迷你版”的 ASP.NET MVC。值得一提的是，为了让读者根据该实例从真正的 ASP.NET MVC 中找到对应的组件，本书完全采用了与 ASP.NET MVC 一致的类型命名方式。

1.4.1 建立在“迷你版” ASP.NET MVC 上的 Web 应用

在正式介绍我们自己创建的“迷你版”ASP.NET MVC 的实现原理之前，不妨来看看建立在该框架之上的 Web 应用如何定义。通过 Visual Studio 创建一个空的 ASP.NET Web 应用（注意不是 ASP.NET MVC 应用）并不会引用 `System.Web.Mvc.dll` 这个程序集，所以在接下来的程序中看到的所谓 MVC 的组件都是我们自行定义的。

首先定义了如下一个 `SimpleModel` 类型，它表示最终需要绑定到 View 上的数据。为了验证针对 `Controller` 和 `Action` 的解析机制，`SimpleModel` 定义的两个属性分别表示当前请求的目标 `Controller` 和 `Action`。

```
public class SimpleModel
{
    public string Controller { get; set; }
    public string Action { get; set; }
}
```

与真正的 ASP.NET MVC 应用开发一样，我们需要定义 `Controller` 类。按照约定的命名方式（以字符“`Controller`”作为后缀），我们定义了如下一个 `HomeController`。`HomeController` 实现的抽象类型 `ControllerBase` 是我们自行定义的。以自定义的 `ActionResult` 作为返回类型的 `Index` 方法表示 `Controller` 的 `Action`，它接受一个 `SimpleModel` 类型的对象作为参数。该 `Action` 方法返回的 `ActionResult` 是一个 `RawContextResult` 对象，顾名思义，`RawContextResult` 就是将指定的内容进行原样返回。在这里我们将作为参数的 `SimpleModel` 对象的 `Controller` 和 `Action` 属性显示出来。

```
public class HomeController: ControllerBase
{
    public ActionResult Index(SimpleModel model)
    {
        string content = string.Format("Controller: {0}<br/>Action:{1}",
            model.Controller, model.Action);
    }
}
```

```

        model.Controller, model.Action);
    return new RawContentResult(content);
}
}

```

ASP.NET MVC 根据请求地址来解析出用于处理该请求的 Controller 的类型和 Action 方法名称。具体来说，我们预注册一些包含 Controller 和 Action 名称作为占位符的（相对）地址模板，如果请求地址符合相应地址模板的模式，Controller 和 Action 名称就可以正确地解析出来。和 ASP.NET MVC 应用类似，我们在 Global.asax 中注册了如下一个地址模板（{controller}/{action}）。我们还注册了一个用于创建 Controller 对象的工厂。RouteTable、ControllerBuilder 和 DefaultControllerFactory 都是我们自定义的类型。

```

public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.Add("default",
            new Route{Url = "{controller}/{action}"});
        ControllerBuilder.Current.SetControllerFactory(
            new DefaultControllerFactory());
    }
}

```

正如上面所说的，ASP.NET MVC 是通过一个自定义的 HttpModule 实现的，在这个“迷你版”ASP.NET MVC 框架中我们也将起名为 UrlRoutingModule。在运行 Web 应用之前，我们需要通过配置对该自定义 HttpModule 进行注册，下面是相关的配置。

```

<configuration>
  <system.webServer>
    <modules>
      <add name="UrlRoutingModule"
          type="WebApp.UrlRoutingModule, WebApp"/>
    </modules>
  </system.webServer>
</configuration>

```

到目前为止，所有的编程和配置工作已经完成，为了让定义在 HomeController 中的 Action 方法 Index 来处理针对该 Web 应用的访问请求，我们需要指定与之匹配的地址（符合定义在注册地址模板的 URL 模式）。如图 1-12 所示，由于在浏览器中输入地址（http://.../Home/Index）正好对应着 HomeController 的 Action 方法 Index，所以对应的方法会被执行，而执行的结果就是将当前请求的目标 Controller 和 Action 的名称显示出来。（S102）

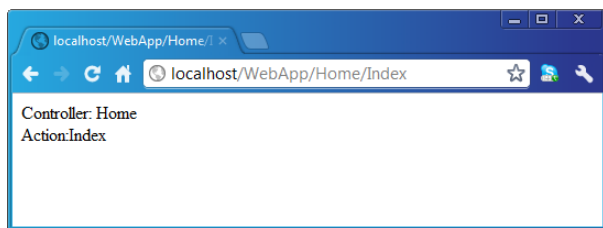


图 1-12 采用符合注册的路由地址模板的地址访问 Web 应用

上面演示了如何在我们自己创建的“迷你版”ASP.NET MVC 框架中创建一个 Web 应用，从中可以看到和创建一个真正的 ASP.NET MVC 应用别无二致。接下来我们就来逐步地分析这个自定义的 ASP.NET MVC 框架是如何建立起来的，而它也代表了真正的 ASP.NET MVC 框架的工作原理。

1.4.2 URL 路由

对于一个 ASP.NET MVC 应用来说，针对 HTTP 请求的处理实现在某个 Controller 类型的某个 Action 方法中，每个 HTTP 请求不再像 ASP.NET Web Forms 应用一样是对应着一个物理文件，而是对应着某个 Controller 的某个 Action。目标 Controller 和 Action 的名称包含在 HTTP 请求的 URL 中，而 ASP.NET MVC 的首要任务就是通过当前 HTTP 请求的解析得到正确的 Controller 和 Action 的名称，这个过程是通过 ASP.NET MVC 的 URL 路由机制来实现的。

RouteData

ASP.NET 定义了一个全局的路由表，路由表中的每个路由对象包含一个 URL 模板。目标 Controller 和 Action 的名称可以通过路由变量以占位符（比如“{controller}”和“{action}”）定义在 URL 模板中，也可以作为路由对象的默认值。对于每一个抵达的 HTTP 请求，ASP.NET MVC 会遍历路由表找到一个具有与当前请求 URL 模式相匹配的路由对象，并最终解析出以 Controller 和 Action 名称为核心的路由数据。在我们自定义的 ASP.NET MVC 框架中，路由数据通过具有如下定义的 RouteData 类型表示。

```
public class RouteData
{
    public IDictionary<string, object> Values { get; private set; }
    public IDictionary<string, object> DataTokens { get; private set; }
    public IRouteHandler RouteHandler { get; set; }
    public RouteBase Route { get; set; }

    public RouteData()
    {
        this.Values = new Dictionary<string, object>();
        this.DataTokens = new Dictionary<string, object>();
        this.DataTokens.Add("namespaces", new List<string>());
    }

    public string Controller
    {
        get
        {
            object controllerName = string.Empty;
            this.Values.TryGetValue("controller", out controllerName);
            return controllerName.ToString();
        }
    }

    public string ActionName
```

```

    {
        get
        {
            object actionName = string.Empty;
            this.Values.TryGetValue("action", out actionName);
            return actionName.ToString();
        }
    }
}

```

如上面的代码片段所示，RouteData 定义了两个字典类型的属性 Values 和 DataTokens，前者代表直接从请求地址解析出来的变量列表，后者代表具有其他来源的变量列表。表示 Controller 和 Action 名称的同名属性直接从 Values 字典中提取，对应的 Key 分别为 controller 和 action。

我们之前已经提到过 ASP.NET MVC 本质上是由两个自定义的 ASP.NET 组件来实现的，一个是自定义的 HttpModule，另一个是自定义的 HttpHandler，而后者从 RouteData 的 RouteHandler 属性获得。RouteData 的 RouteHandler 属性类型为 IRouteHandler 接口，如下面的代码片段所示，该接口具有一个唯一的 GetHttpHandler 用于返回真正用于处理 HTTP 请求的 HttpHandler 对象。

```

public interface IRouteHandler
{
    IHttpHandler GetHttpHandler(RequestContext requestContext);
}

```

IRouteHandler 接口的 GetHttpHandler 方法接受一个类型为 RequestContext 的参数，顾名思义，RequestContext 表示当前（HTTP）请求的上下文，其核心就是对当前 HttpContext 和 RouteData 的封装，这可以通过如下的代码片段看出来。

```

public class RequestContext
{
    public virtual HttpContextBase HttpContext { get; set; }
    public virtual RouteData RouteData { get; set; }
}

```

Route 和 RouteTable

RouteData 具有一个类型为 RouteBase 的 Route 属性，表示生成路由数据对应的路由对象。如下面的代码片段所示，RouteBase 是一个抽象类，它仅仅包含一个 GetRouteData 方法。该方法判断是否与当前请求相匹配，并在匹配的情况下返回用于封装路由数据的 RouteData 对象。该方法接受一个表示当前 HTTP 上下文的 HttpContextBase 对象，如果与当前请求不匹配，则返回 Null。

```

public abstract class RouteBase
{
    public abstract RouteData GetRouteData(HttpContextBase httpContext);
}

```

ASP.NET MVC 提供的基于 URL 模板的路由机制是通过其子类 `Route` 实现的。如下面的代码片段所示，它具有一个代表 URL 模板的字符串类型的 `Url` 属性。在实现的 `GetRouteData` 方法中，我们通过 `HttpContextBase` 获取当前请求的 URL，如果它与 URL 模板的模式相匹配则创建一个 `RouteData` 返回，否则返回 `Null`。对于返回的 `RouteData` 对象，其 `Values` 属性表示的字典对象包含直接通过地址解析出来的变量，而对于 `DataTokens` 字典和 `RouteHandler` 属性，则直接取自 `Route` 对象的同名属性。

```
public class Route : RouteBase
{
    public IRouteHandler                RouteHandler { get; set; }
    public string                       Url { get; set; }
    public IDictionary<string, object> DataTokens { get; set; }

    public Route()
    {
        this.DataTokens      = new Dictionary<string, object>();
        this.RouteHandler    = new MvcRouteHandler();
    }

    public override RouteData GetRouteData(HttpContextBase httpContext)
    {
        IDictionary<string, object> variables;
        if (this.Match(httpContext.Request
            .AppRelativeCurrentExecutionFilePath.Substring(2), out variables))
        {
            RouteData routeData = new RouteData();
            foreach (var item in variables)
            {
                routeData.Values.Add(item.Key, item.Value);
            }
            foreach (var item in DataTokens)
            {
                routeData.DataTokens.Add(item.Key, item.Value);
            }
            routeData.RouteHandler = this.RouteHandler;
            return routeData;
        }
        return null;
    }

    protected bool Match(string requestUrl,
        out IDictionary<string, object> variables)
    {
        variables      = new Dictionary<string, object>();
        string[] strArray1    = requestUrl.Split('/');
        string[] strArray2    = this.Url.Split('/');
        if (strArray1.Length != strArray2.Length)
        {
            return false;
        }
        for (int i = 0; i < strArray2.Length; i++)
        {
            if (strArray2[i].StartsWith("{") && strArray2[i].EndsWith("}"))
            {

```



```

        variables.Add(strArray2[i].Trim("{}").ToCharArray(), strArray1[i]);
    }
}
return true;
}
}

```

由于同一个 Web 应用可以采用多种不同的 URL 模式，所以需要注册多个继承自 `RouteBase` 的路由对象，多个路由对象组成了一个路由表。在我们自定义 ASP.NET MVC 框架中，路由表通过类型 `RouteTable` 表示。如下面的代码片段所示，`RouteTable` 仅仅具有一个类型为 `RouteDictionary` 的 `Routes` 属性表示针对整个 Web 应用的全局路由表。

```

public class RouteTable
{
    public static RouteDictionary Routes { get; private set; }
    static RouteTable()
    {
        Routes = new RouteDictionary();
    }
}

```

`RouteDictionary` 表示一个具名的路由对象的列表，我们直接让它继承自泛型的字典类型 `Dictionary<string, RouteBase>`，其中的 Key 表示路由对象的注册名称。在 `GetRouteData` 方法中，我们遍历集合找到与指定的 `HttpContextBase` 对象匹配的路由对象，并得到对应的 `RouteData`。

```

public class RouteDictionary: Dictionary<string, RouteBase>
{
    public RouteData GetRouteData(HttpContextBase httpContext)
    {
        foreach (var route in this.Values)
        {
            RouteData routeData = route.GetRouteData(httpContext);
            if (null != routeData)
            {
                return routeData;
            }
        }
        return null;
    }
}

```

在 `Global.asax` 中我们创建了一个基于指定 URL 模板 (“{controller}/{action}”) 的 `Route` 对象，并将其添加到通过 `RouteTable` 的静态只读属性 `Routes` 所表示的全局路由表中。

UrlRoutingModule

路由表的作用是对当前的 HTTP 请求的 URL 进行解析，从而获取一个以 Controller 和 Action 名称为核心的路由数据，即上面介绍的 `RouteData` 对象。整个解析工作是通过一个类型为 `UrlRoutingModule` 的自定义 `HttpModule` 来完成的。如下面的代码片段所示，在实现了接口 `IHttpModule` 的 `UrlRoutingModule` 类型的 `Init` 方法中，我们注册了 `HttpApplicataion` 的

PostResolveRequestCache 事件。

```
public class UrlRoutingModule: IHttpModule
{
    public void Dispose()
    {}

    public void Init(HttpApplication context)
    {
        context.PostResolveRequestCache += OnPostResolveRequestCache;
    }
    protected virtual void OnPostResolveRequestCache(object sender, EventArgs e)
    {
        HttpContextWrapper httpContext =
            new HttpContextWrapper(HttpContext.Current);
        RouteData routeData = RouteTable.Routes.GetRouteData(httpContext);
        if (null == routeData)
        {
            return;
        }
        RequestContext requestContext = new RequestContext {
            RouteData = routeData, HttpContext = httpContext };
        IHttpHandler handler =
            routeData.RouteHandler.GetHttpHandler(requestContext);
        httpContext.RemapHandler(handler);
    }
}
```

当 PostResolveRequestCache 事件触发之后, UrlRoutingModule 通过 RouteTable 的静态只读属性 Routes 得到表示全局路由表的 RouteDictionary 对象, 然后调用其 GetRouteData 方法并传入用于封装当前 HttpContext 的 HttpContextWrapper 对象 (HttpContextWrapper 是 HttpContextBase 的子类) 最终得到一个封装路由数据的 RouteData 对象。如果得到的 RouteData 不为 Null, 则根据该对象本身和之前得到的 HttpContextWrapper 对象创建一个表示当前请求上下文的 RequestContext 对象, 将其作为参数传入 RouteData 的 RouteHandler 的 GetHttpHandler 方法得到一个 IHttpHandler 对象。最后我们调用 HttpContextWrapper 对象的 RemapHandler 方法将得到的 IHttpHandler 进行映射使之用于对当前 HTTP 请求的处理。

1.4.3 Controller 的激活

ASP.NET MVC 的 URL 路由系统通过注册的路由表对 HTTP 请求进行解析从而得到一个用于封装路由数据的 RouteData 对象, 而这个过程是通过自定义的 UrlRoutingModule 对 HttpApplication 的 PostResolveRequestCache 事件进行注册实现的。RouteData 中已经包含了目标 Controller 的名称, 我们需要根据该名称激活对应的 Controller 对象。接下来进一步分析真正的 Controller 对象是如何被激活的。

MvcRouteHandler

通过前面的介绍我们知道, 继承自 RouteBase 的 Route 类型具有一个类型为

IRouteHandler 接口的属性 RouteHandler，它主要的用途就是用于根据指定的请求上下文（通过一个 RequestContext 对象表示）来获取一个 HttpHandler 对象。当 GetRouteData 方法被执行后，Route 的 RouteHandler 属性值将反映在得到的 RouteData 的同名属性上。在默认的情况下，Route 的 RouteHandler 属性是一个 MvcRouteHandler 对象，如下的代码片段反映了这一点。

```
public class Route : RouteBase
{
    //其他成员
    public IRouteHandler RouteHandler { get; set; }
    public Route()
    {
        //其他操作
        this.RouteHandler = new MvcRouteHandler();
    }
}
```

对于我们这个“迷你版”的 ASP.NET MVC 框架来说，MvcRouteHandler 是一个具有如下定义的类型，在实现的 GetHttpHandler 方法中，它会直接返回一个 MvcHandler 对象。

```
public class MvcRouteHandler: IRouteHandler
{
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        return new MvcHandler(requestContext);
    }
}
```

MvcHandler

在前面的内容中已经提到整个 ASP.NET MVC 框架是通过自定义的 HttpModule 和 HttpHandler 对象 ASP.NET 进行扩展实现的。这个自定义 HttpModule 已经介绍过了，就是 UrlRoutingModule，而这个自定义的 HttpHandler 则是要重点介绍的 MvcHandler。

UrlRoutingModule 在通过路由表解析 HTTP 请求得到一个用于封装路由数据的 RouteData 后，会调用其 RouteHandler 的 GetHttpHandler 方法得到 HttpHandler 对象并注册到当前的 HTTP 上下文。由于 RouteData 的 RouteHandler 来源于对应 Route 对象的 RouteHandler，而后者在默认的情况下是一个 MvcRouteHandler 对象，所以默认情况下用于处理 HTTP 请求的就是这么一个 MvcHandler 对象。MvcHandler 实现了对 Controller 对象的激活和对相应 Action 方法的执行。

下面的代码片段体现了整个 MvcHandler 的定义，它具有一个类型为 RequestContext 的属性，表示被处理的当前请求上下文，该属性在构造函数中指定。在实现的 ProcessRequest 中实现了对 Controller 对象的激活和执行。

```
public class MvcHandler: IHttpHandler
{
    public bool IsReusable
```

```

    {
        get{return false;}
    }
    public RequestContext RequestContext { get; private set; }

    public MvcHandler(RequestContext requestContext)
    {
        this.RequestContext = requestContext;
    }

    public void ProcessRequest(HttpContext context)
    {
        string controllerName = this.RequestContext.RouteData.Controller;
        IControllerFactory controllerFactory =
            ControllerBuilder.Current.GetControllerFactory();
        IController controller = controllerFactory.CreateController(
            this.RequestContext, controllerName);
        controller.Execute(this.RequestContext);
    }
}

```

Controller 与 ContrllerFactory

我们为 Controller 定义了一个接口 `Icontroller`，如下面的代码片段所示，该接口具有唯一的方法 `Execute` 表示对 Controller 的执行。该方法在 `MvcHandler` 的 `ProcessRequest` 方法中被执行，而传入该方法的参数是表示当前请求上下文的 `RequestContext` 对象。

```

public interface IController
{
    void Execute(RequestContext requestContext);
}

```

从 `MvcHandler` 的定义可以看到 `Controller` 对象的激活是通过工厂模式实现的，我们为 `Controller` 工厂定义了一个具有如下定义的 `IControllerFactory` 接口。`IControllerFactory` 通过 `CreateController` 方法根据传入的请求上下文和 `Controller` 的名称来激活相应的 `Controller` 对象。

```

public interface IControllerFactory
{
    IController CreateController(RequestContext requestContext,
        string controllerName);
}

```

在 `MvcHandler` 的 `ProcessRequest` 方法中，它通过 `ControllerBuilder` 的静态属性 `Current` 得到当前的 `ControllerBuilder` 对象，并调用 `GetControllerFactory` 方法获得当前的 `ControllerFactory`。然后通过从 `RequestContext` 中提取的 `RouteData` 获得 `Controller` 的名称，最后将它连同 `RequestContext` 一起作为参数调用 `ContollerFactory` 的 `CreateController` 方法实现对目标 `Controller` 对象的创建。

`ControllerBuilder` 的整个定义如下面的代码片段所示，表示当前 `ControllerBuilder` 的静态只读属性的 `Current` 在静态构造函数中被创建。`SetControllerFactory` 和 `GetControllerFactory` 方法用于 `ContorllerFactory` 的注册和获取。

```

public class ControllerBuilder
{
    private Func<IControllerFactory> factoryThunk;
    public static ControllerBuilder Current { get; private set; }

    static ControllerBuilder()
    {
        Current = new ControllerBuilder();
    }

    public IControllerFactory GetControllerFactory()
    {
        return factoryThunk();
    }

    public void SetControllerFactory(IControllerFactory controllerFactory)
    {
        factoryThunk = () => controllerFactory;
    }
}

```

再回头看看之前建立在自定义 ASP.NET MVC 框架的 Web 应用，我们就是通过当前的 **ControllerBuilder** 来注册 **ControllerFactory**。如下面的代码片段所示，注册的 **ControllerFactory** 的类型为 **DefaultControllerFactory**。

```

public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        //其他操作
        ControllerBuilder.Current.SetControllerFactory(
            new DefaultControllerFactory());
    }
}

```

作为默认 **ControllerFactory** 的 **DefaultControllerFactory** 类型定义如下。激活 **Controller** 对象的前提是能够正确解析出 **Controller** 的真实类型，作为 **CreateController** 方法输入参数的 **controllerName** 仅仅表示 **Controller** 的名称，我们需要加上 **Controller** 字符后缀作为类型名称。在 **DefaultControllerFactory** 类型被加载的时候（静态构造函数被调用），通过 **BuildManager** 加载所有引用的程序集，并得到所有实现了接口 **IController** 的类型并将其缓存起来。在 **CreateController** 方法中根据 **Controller** 的名称和命名空间从保存的 **Controller** 类型列表中得到对应的 **Controller** 类型，并通过反射的方式创建它。

```

public class DefaultControllerFactory : IControllerFactory
{
    private static List<Type> controllerTypes = new List<Type>();

    static DefaultControllerFactory()
    {
        foreach (Assembly assembly in BuildManager.GetReferencedAssemblies())
        {

```

```

        foreach (Type type in assembly.GetTypes().Where(
            type => typeof(HomeController).IsAssignableFrom(type)))
        {
            controllerTypes.Add(type);
        }
    }

    public HomeController CreateController(RequestContext requestContext,
        string controllerName)
    {
        string typeName = controllerName + "Controller";
        Type controllerType = controllerTypes.FirstOrDefault(
            c => string.Compare(typeName, c.Name, true) == 0);
        if (null == controllerType)
        {
            return null;
        }
        return (HomeController)Activator.CreateInstance(controllerType);
    }
}

```

上面我们详细地介绍了 **Controller** 的激活原理，现在将关注点返回到 **Controller** 自身。通过实现 **HomeController** 接口我们为所有的 **Controller** 定义了一个具有如下定义的 **ControllerBase** 抽象基类，从中可以看到在实现的 **Execute** 方法中，**ControllerBase** 通过一个实现了接口 **IActionInvoker** 的对象完成了针对 **Action** 方法的执行。

```

public abstract class ControllerBase: HomeController
{
    protected IActionInvoker ActionInvoker { get; set; }

    public ControllerBase()
    {
        this.ActionInvoker = new ControllerActionInvoker();
    }

    public void Execute(RequestContext requestContext)
    {
        ControllerContext context = new ControllerContext {
            RequestContext = requestContext, Controller = this };
        string actionName = requestContext.RouteData.ActionName;
        this.ActionInvoker.InvokeAction(context, actionName);
    }
}

```

1.4.4 Action 的执行

作为 **Controller** 基类 **ControllerBase** 的 **Execute** 方法的核心在于对 **Action** 方法本身的执行和作为方法返回的 **ActionResult** 的执行，两者的执行是通过一个叫做 **ActionInvoker** 的组件来完成的。

ActionInvoker

同样为 **ActionInvoker** 定义了一个接口 **IActionInvoker**，如下面的代码片段所示，该接口定义了一个唯一的方法 **InvokeAction** 用于执行指定名称的 **Action** 方法，该方法的第一个参数是一个表示基于当前 **Controller** 上下文的 **ControllerContext** 对象。

```
public interface IActionInvoker
{
    void InvokeAction(ControllerContext controllerContext, string actionName);
}
```

ControllerContext 类型在真正的 ASP.NET MVC 框架中要复杂一些，在这里我们对它进行了简化，仅仅将它表示成对当前 **Controller** 和请求上下文的封装，而这两个要素分别通过如下所示的 **Controller** 和 **RequestContext** 属性表示。

```
public class ControllerContext
{
    public ControllerBase Controller { get; set; }
    public RequestContext RequestContext { get; set; }
}
```

ControllerBase 中表示 **ActionInvoker** 的同名属性在构造函数中被初始化。在 **Execute** 方法中，通过作为方法参数的 **RequestContext** 对象创建 **ControllerContext** 对象，并通过包含在 **RequestContext** 中的 **RouteData** 得到目标 **Action** 的名称，然后将这两者作为参数调用 **ActionInvoker** 的 **InvokeAction** 方法。

从前面给出的关于 **ControllerBase** 的定义中可以看到在构造函数中默认创建的 **ActionInvoker** 是一个类型为 **ControllerActionInvoker** 的对象。如下所示的代码片段反映了整个 **ControllerActionInvoker** 的定义，**InvokeAction** 方法的目的在于实现针对 **Action** 方法的执行。由于 **Action** 方法具有相应的参数，在执行 **Action** 方法之前必须进行参数的绑定。ASP.NET MVC 将这个机制称为 **Model** 的绑定，而这又涉及另一个重要的组件 **ModelBinder**。

```
public class ControllerActionInvoker : IActionInvoker
{
    public IModelBinder ModelBinder { get; private set; }

    public ControllerActionInvoker()
    {
        this.ModelBinder = new DefaultModelBinder();
    }

    public void InvokeAction(ControllerContext controllerContext,
        string actionName)
    {
        MethodInfo method = controllerContext.Controller.GetType().GetMethods()
            .First(m => string.Compare(actionName, m.Name, true) == 0);
        List<object> parameters = new List<object>();
        foreach (ParameterInfo parameter in method.GetParameters())
        {
            parameters.Add(this.ModelBinder.BindModel(controllerContext,
                parameter.Name, parameter.ParameterType));
        }
    }
}
```

```

    }
    ActionResult actionResult = method.Invoke(controllerContext.Controller,
        parameters.ToArray()) as ActionResult;
    actionResult.ExecuteResult(controllerContext);
}
}

```

ModelBinder

我们为 **ModelBinder** 提供了一个简单的定义，这与在真正的 ASP.NET MVC 中的同名接口的定义不尽相同。如下面的代码片段所示，该接口具有唯一的 **BindModel** 方法，根据 **ControllerContext** 和 **Model** 名称（在这里实际上是参数名称）和类型得到一个作为参数的对象。

```

public interface IModelBinder
{
    object BindModel(ControllerContext controllerContext, string modelName,
        Type modelType);
}

```

通过前面给出的关于 **ControllerActionInvoker** 的定义可以看到，在构造函数中默认创建的 **ModelBinder** 对象是一个 **DefaultModelBinder** 对象，由于仅仅是对 ASP.NET MVC 的模拟，定义在自定义的 **DefaultModelBinder** 中的 **Model** 绑定逻辑比 ASP.NET MVC 的 **DefaultModelBinder** 要简单得多，很多复杂的 **Model** 机制并未在我们自定义的 **DefaultModelBinder** 体现出来。

如下面的代码片段所示，绑定到参数上的数据具有三个来源：**HTTP-POST Form**、**RouteData** 的 **Values** 和 **DataTokens** 属性，它们都是字典结构的数据集合。如果参数类型为字符串或者简单的值类型，我们可以直接根据参数名称和 **Key** 进行匹配；对于复杂类型（比如之前例子中定义的包含 **Controller** 和 **Action** 名称的数据类型 **SimpleModel**），则通过反射根据类型创建新的对象，并根据属性名称与 **Key** 的匹配关系对相应的属性进行赋值。

```

public class DefaultModelBinder : IModelBinder
{
    public object BindModel(ControllerContext controllerContext,
        string modelName, Type modelType)
    {
        if (modelType.IsValueType || typeof(string) == modelType)
        {
            object instance;
            if (GetValueTypeInstance(controllerContext, modelName,
                modelType, out instance))
            {
                return instance;
            };
            return Activator.CreateInstance(modelType);
        }

        object modelInstance = Activator.CreateInstance(modelType);
        foreach (PropertyInfo property in modelType.GetProperties())
        {

```



```

        if (!property.CanWrite || (!property.PropertyType.IsValueType
            &&property.PropertyType!=typeof(string)))
        {
            continue;
        }
        objectpropertyValue;
        if (GetValueTypeInstance(controllerContext, property.Name,
            property.PropertyType, out propertyValue))
        {
            property.SetValue(modelInstance, propertyValue, null);
        }
    }
    returnmodelInstance;
}

private boolGetValueTypeInstance(ControllerContext controllerContext,
    stringmodelName, Type modelType, out object value)
{
    var form = HttpContext.Current.Request.Form;
    string key;
    if (null != form)
    {
        key = form.AllKeys.FirstOrDefault(k =>string.Compare(k,
            modelName, true) == 0);
        if (key != null)
        {
            value = Convert.ChangeType(form[key], modelType);
            return true;
        }
    }

    key = controllerContext.RequestContext.RouteData.Values
        .Where(item =>string.Compare(item.Key, modelName, true) == 0)
        .Select(item =>item.Key).FirstOrDefault();
    if (null != key)
    {
        value = Convert.ChangeType(controllerContext.RequestContext
            .RouteData.Values[key], modelType);
        return true;
    }

    key = controllerContext.RequestContext.RouteData.DataTokens
        .Where(item =>string.Compare(item.Key, modelName, true) == 0)
        .Select(item =>item.Key).FirstOrDefault();
    if (null != key)
    {
        value = Convert.ChangeType(controllerContext.RequestContext
            .RouteData.DataTokens[key], modelType);
        return true;
    }
    value = null;
    return false;
}
}

```

在 `ControllerActionInvoker` 的 `InvokeAction` 方法中，我们直接将传入的 `Action` 名称作为方法名从 `Controller` 类型中得到表示 `Action` 操作的 `MethodInfo` 对象，然后遍历 `MethodInfo` 的参数列表，对于每一个 `ParameterInfo` 对象，我们将它的 `Name` 和 `ParameterType` 属性表示

的参数名称和类型，连同创建的 `ControllerContext` 作为参数调用 `ModelBinder` 的 `BindModel` 方法并得到对应的参数值，最后通过反射的方式传入参数列表并执行 `MethodInfo`。

和真正的 ASP.NET MVC 一样，定义在 `Controller` 的 `Action` 方法返回一个 `ActionResult` 对象，我们通过执行它的 `Execute` 方法实现对请求的响应。

ActionResult

我们为具体的 `ActionResult` 定义了一个 `ActionResult` 抽象基类，如下面的代码片段所示，该抽象类具有一个参数类型为 `ControllerContext` 的抽象方法 `ExecuteResult`，我们最终对请求的响应就实现在该方法中。

```
public abstract class ActionResult
{
    public abstract void ExecuteResult(ControllerContext context);
}
```

在之前创建的例子中，`Action` 方法返回的是一个类型为 `RawContentResult` 的对象，顾名思义，`RawContentResult` 将初始化时指定的内容（字符串）原封不动地写入针对当前请求的 HTTP 响应消息中，具体的实现如下所示。

```
public class RawContentResult: ActionResult
{
    public string RawData { get; private set; }
    public RawContentResult(string rawData)
    {
        RawData = rawData;
    }
    public override void ExecuteResult(ControllerContext context)
    {
        context.RequestContext.HttpContext.Response.Write(this.RawData);
    }
}
```

本章小结

ASP.NET MVC 是在现有的 ASP.NET 平台上基于 MVC 架构模式创建的 Web 应用开发框架。MVC 体现了界面呈现、UI 处理逻辑和业务逻辑之间的分离，传统的 MVC 并没有对 Model、View 和 Controller 之间的交互进行严格的约束。在软件设计的发展历程中出现了一些 MVC 的变体，它们遵循定义在 MVC 中的基本原则，并定义更加严格的交互规则，其中 MVP 和 Model 2 是两个典型的 MVC 变体，而 ASP.NET MVC 就是对 Model 2 的实现。

ASP.NET 采用极具扩展性的管道式设计，`HttpApplication` 是整个 ASP.NET 管道的核心，它定义了一系列的事件，它们会在请求处理过程中相应的阶段被触发。`HttpModule` 是成就 ASP.NET 可扩展的“头号功臣”，通过 `HttpModel` 注册 `HttpApplication` 相应的事件帮助我们

在某个阶段参与到对请求处理的整个流程之中，而请求的最终处理者是注册的 `HttpHandler`。
ASP.NET MVC 实际上是通过自定义的 `HttpModule` 和 `HttpHandler` 构建的。

为了让读者对 ASP.NET MVC 对从“接收请求”到“回复响应”的整个处理流程有一个大致的了解，我们按照 ASP.NET MVC 本身的实现原理构建了一个模拟程序，该程序模拟了 URL 路由、Controller 的激活、Action 的执行和 View 的呈现，可以将此模拟程序看成是一个“迷你版”的 ASP.NET MVC。

第2章 URL 路由

HttpModule 和 HttpHandler 是 ASP.NET 管道的两个重要的组件。请求的最终处理通过 Handler 来完成,ASP.NET MVC 就是通过一个名为 MvcHandler 的自定义 HttpHandler 实现了对 Controller 的激活和 Action 的执行。但是在这之前对 Controller 和 Action 名称的解析则是通过 ASP.NET 的 URL 路由系统来完成的,而整个 URL 路由系统是通过一个名为 UrlRoutingModule 的自定义 HttpModule 实现的。

2.1 ASP.NET 路由系统

ASP.NET MVC 对请求的处理最终体现在对激活的目标 Controller 对应的 Action 方法的执行。一般来说, 目标 Controller 和 Action 的名称由请求的 URL 决定, URL 路由系统通过对请求的拦截和对请求 URL 的解析, 得到以 Controller 和 Action 名称为核心的路由数据。URL 路由系统并不是专属于 ASP.NET MVC 的, 而是直接建立在 ASP.NET 上。ASP.NET 通过 URL 路由系统实现了请求地址与物理文件的分离。

2.1.1 请求 URL 与物理文件的分离

对于一个 ASP.NET Web Forms 应用来说, 每一个有效的请求都对应着一个具体的物理文件。部署在 Web 服务器上的物理文件可以是静态的 (比如图片和静态 HTML 文件等), 也可以是动态的 (比如.aspx 文件)。对于静态文件的请求, ASP.NET 直接返回文件的整个内容, 而针对动态文件的请求则会涉及到相关代码的执行。但是这种将 URL 与物理文件紧密绑定在一起的方式并不是一种好的解决方案, 它带来的局限性主要体现在如下几个方面。

- **灵活性:** 由于 URL 是对物理文件路径的反映, 意味着如果物理文件的路径发生了改变 (比如改变了文件的目录结构或者文件名), 原来基于该文件的链接将变得无效。
- **可读性:** 在很多情况下, URL 不仅仅具备基本的可用性 (能够访问正确的网络资源), 还需要具有很好的可读性。好的 URL 设计应该让我们一眼就能看出针对它访问的目标资源是什么。请求地址与物理文件紧密绑定会让我们完全失去了定义可读性 URL 的机会。
- **SEO 优化:** 对于网站开发来说, 为了迎合搜索引擎检索的规则, 我们需要对 URL 进行有效的设计使之能易于被主流的引擎检索收录, 如果 URL 完全与物理地址关联, 这无异于失去了 SEO 优化的能力。

我们需要一种更加灵活的机制来实现请求地址与文件路径的分离。说到这里, 可能很多人会想到 URL 重写。为了使 Web 应用可以独立地设计用于访问应用资源的 URL, 微软为 IIS 7 编写了一个 URL 重写模块。这是一个基于规则的 URL 重写引擎, 它在 URL 被 Web 服务器处理之前根据定义的规则重定向某个物理文件。

URL 重写在 IIS 级别解决了 URL 与物理地址的分离, 它通过一个基于本地 (Native) 代码的模块注册到 IIS 管道上, 所以可以应用于所有寄宿于 IIS 中的 Web 应用, 而 URL 路由系统则是 ASP.NET 的一部分, 是通过托管代码实现的。为了让读者对 ASP.NET 的 URL 路由具有一个感官的认识, 我们来演示一个简单的实例。

2.1.2 实例演示：通过 URL 路由实现请求地址与.aspx 页面的映射（S201）

我们创建一个简单的 ASP.NET Web Forms 应用，并采用一个独立于.aspx 文件路径的 URL 来访问对应的 Web 页面，两者之间的映射通过 URL 路由来实现。我们依然沿用第 1 章关于员工管理的场景，可以创建一个页面来显示员工的列表和某个员工的详细信息，呈现效果如图 2-1 所示。

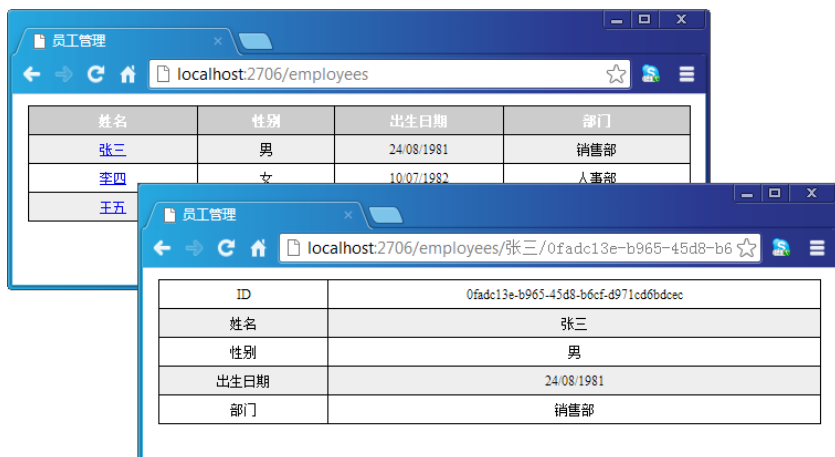


图 2-1 员工列表和员工详细信息页面

我们将关注点放到如图 2-1 所示的两个页面的 URL 上，用于显示员工列表的页面地址为 `http://.../employees`，当用户点击某个显示为姓名的链接后，用于显示所选员工详细信息的页面被呈现出来，其页面地址的 URL 模式为“`http://.../employees/{姓名}/{ID}`”。对于后者，最终用户一眼可以从 URL 中看出通过该地址获取的是哪个员工的信息。

有人可能会问，为什么我们要在 URL 中同时包含员工的姓名和 ID 呢？这是因为 ID（本例采用 GUID）的可读性不如员工姓名，但是员工姓名不具有唯一性，在这里我们使用的 ID 是为了逻辑处理的需要而提供的唯一标识，而姓名则是出于可读性的需要。

我们将员工的所有信息（ID、姓名、性别、出生日期和所在部门）定义在如下所示的 `Employee` 类型中，它与我们在第 1 章“ASP.NET + MVC”中演示 Model2 模式中的同名类型具有一致的定义。我们照例定义了如下一个 `EmployeeRepository` 类型来维护员工列表的数据。简单起见，员工列表通过静态字段 `employees` 表示。`EmployeeRepository` 的 `GetEmployees` 方法根据指定的 ID 返回包含指定的员工，如果指定的 ID 为“*”，则返回所有员工列表。

```
public class Employee
{
    public string      Id { get; private set; }
    public string      Name { get; private set; }
}
```

```

public string      Gender { get; private set; }
public DateTime    BirthDate { get; private set; }
public string      Department { get; private set; }

public Employee(string id, string name, string gender, DateTime birthDate,
    string department)
{
    this.Id          = id;
    this.Name        = name;
    this.Gender       = gender;
    this.BirthDate    = birthDate;
    this.Department   = department;
}
}

public class EmployeeRepository
{
    private static IList<Employee> employees;
    static EmployeeRepository()
    {
        employees = new List<Employee>();
        employees.Add(new Employee(Guid.NewGuid().ToString(), "张三", "男",
            new DateTime(1981, 8, 24), "销售部"));
        employees.Add(new Employee(Guid.NewGuid().ToString(), "李四", "女",
            new DateTime(1982, 7, 10), "人事部"));
        employees.Add(new Employee(Guid.NewGuid().ToString(), "王五", "男",
            new DateTime(1981, 9, 21), "人事部"));
    }
    public IEnumerable<Employee> GetEmployees(string id = "")
    {
        return employees.Where(e => e.Id == id || string.IsNullOrEmpty(id) ||
            id=="*");
    }
}

```

对于如图 2-1 所示的两个页面实际上对应着同一个.aspx 文件，即作为 Web 应用默认页面的 Default.aspx。要通过一个独立于物理路径的 URL 来访问该.aspx 页面，就需要采用 URL 路由机制来实现两者之间的映射。我们将实现映射的路由注册代码定义在 Global.asax 文件中，如下面的代码片段所示，在 Application_Start 方法中通过 System.Web.Routing.RouteTable 的 Routes 属性得到了表示路由对象列表的 System.Web.Routing.RouteCollection 对象，并调用该列表对象的 MapPageRoute 方法将 Default.aspx 页面（~/Default.aspx）与一个 URL 模板（employees/{name}/{id}）进行了映射。

```

public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        var defaults = new RouteValueDictionary{{"name","*"}, {"id","*"} };
        RouteTable.Routes.MapPageRoute("", "employees/{name}/{id}",
            "~/Default.aspx", true, defaults);
    }
}

```

作为 MapPageRoute 方法最后一个参数的 RouteValueDictionary 对象用于指定定义在路由

模板中路由变量 (“{name}” 和 “{id}”) 的默认值。如果我们为定义 URL 模板中的路由变量指定了默认值 (指定默认值的变量不一定需要定义在 URL 模板中), 在当前请求地址的后续部分缺失的情况下, 它会采用提供的默认值对该地址进行填充之后再行模式匹配。在如上所示的代码片段中, 我们将 {name} 和 {id} 两变量的默认值均指定为 “*”。对于针对 URL 为 “/employees” 的请求, 我们注册的路由对象会将其格式成 “/employees/*/”, 后者无疑是与定义的 URL 模板模式相匹配的。

在 Default.aspx 页面中, 我们分别采用 GridView 和 DetailsView 来显示所有员工列表和某个列表的详细信息, 下面的代码片段表示该页面主体部分的 HTML。GridView 模板中显示为员工姓名的 HyperLinkField 的链接采用了上面我们定义在 URL 模板 (employees/{name}/{id}) 中的模式。

```
<form id="form1" runat="server">
  <div id="page">
    <asp:GridView ID="GridViewEmployees"
      runat="server" AutoGenerateColumns="false" Width="100%">
      <Columns>
        <asp:HyperLinkField HeaderText="姓名" DataTextField="Name"
          DataNavigateUrlFields="Name, Id"
          DataNavigateUrlFormatString="~/employees/{0}/{1}" />
        <asp:BoundField DataField="Gender" HeaderText="性别" />
        <asp:BoundField DataField="BirthDate" HeaderText="出生日期"
          DataFormatString="{0:dd/MM/yyyy}" />
        <asp:BoundField DataField="Department" HeaderText="部门" />
      </Columns>
    </asp:GridView>
    <asp:DetailsView ID="DetailsViewEmployee" runat="server"
      AutoGenerateRows="false" Width="100%">
      <Fields>
        <asp:BoundField DataField="ID" HeaderText="ID" />
        <asp:BoundField DataField="Name" HeaderText="姓名" />
        <asp:BoundField DataField="Gender" HeaderText="性别" />
        <asp:BoundField DataField="BirthDate" HeaderText="出生日期"
          DataFormatString="{0:dd/MM/yyyy}" />
        <asp:BoundField DataField="Department" HeaderText="部门" />
      </Fields>
    </asp:DetailsView>
  </div>
</form>
```

由于所有员工列表和单一员工的详细信息均体现在该页面中, 所以需要根据其请求地址来判断应该呈现怎样的数据, 而这是通过 RouteData 属性表示的路由数据来实现的。Page 具有一个类型为 System.Web.Routing.RouteData 的 RouteData 属性表示通过 URL 路由系统对当前请求进行解析得到的路由数据。RouteData 的 Values 属性是一个存储路由变量的字典, 其 Key 为变量名称。在如下所示的代码片段中, 我们得到表示员工 ID 的路由变量 (RouteData.Values["id"]), 如果它是默认值 (“*”), 表示当前请求是针对员工列表的, 反之则是针对指定的某个具体员工的。Default.aspx 页面的整个后台代码定义如下。

```

public partial class Default : Page
{
    private EmployeeRepository repository;

    public EmployeeRepository Repository
    {
        get { return null == repository ?
            repository = new EmployeeRepository() : repository; }
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        if (this.IsPostBack)
        {
            return;
        }
        string employeeId = this.RouteData.Values["id"] as string;
        if (employeeId == "*" || string.IsNullOrEmpty(employeeId))
        {
            this.GridViewEmployees.DataSource = this.Repository.GetEmployees();
            this.GridViewEmployees.DataBind();
            this.DetailsViewEmployee.Visible = false;
        }
        else
        {
            var employees = this.Repository.GetEmployees(employeeId);
            this.DetailsViewEmployee.DataSource = employees;
            this.DetailsViewEmployee.DataBind();
            this.GridViewEmployees.Visible = false;
        }
    }
}

```

2.1.3 Route 与 RouteTable

一个 Web 应用具有一个全局的路由表，该路由表通过 `System.Web.Routing.RouteTable` 的静态只读属性 `Routes` 表示，该属性返回一个类型为 `System.Web.Routing.RouteCollection` 的集合。通过上面的实例演示可以看到我们通过调用 `RouteCollection` 的 `MapPageRoute` 方法将某个物理文件路径映射到一个 URL 模板上，这个过程的本质的基于指定的 URL 模板创建一个路由对象并添加到这个全局路由表中。

RouteBase、RouteData 与 VirtualPathData

在 ASP.NET URL 路由系统的应用编程接口中，一个路由对象实现了具有如下定义的抽象类 `System.Web.Routing.RouteBase`。抽象方法 `GetRouteData` 和 `GetVirtualPath` 方法都会试着根据 URL 模板的模式与代表请求地址的 URL 进行匹配，如果匹配失败直接返回 `Null`。在匹配成功的情况下，`GetRouteData` 会得到一个用于封装路由信息的 `RouteData` 对象，而 `GetVirtualPath` 则会生成一个 URL，该 URL 被封装成 `System.Web.Routing.VirtualPathData` 对象返回。

```
public abstract class RouteBase
{
    public abstract RouteData GetRouteData(HttpContextBase httpContext);
    public abstract VirtualPathData GetVirtualPath(RequestContext requestContext,
        RouteValueDictionary values);
}
```

我们现在来看看用于封装路由数据同时作为 `GetRouteData` 方法返回类型的 `RouteData` 具有怎样的定义。如下面的代码片段所示，`RouteData` 具有一个类型为 `Route` 的可读写属性并返回对应的路由对象。

```
public class RouteData
{
    public RouteData();
    public RouteData(RouteBase route, IRouteHandler routeHandler);
    public string GetRequiredString(string valueName);

    public RouteBase                Route {get; set; }
    public IRouteHandler             RouteHandler {get; set; }
    public RouteValueDictionary      DataTokens { get; }
    public RouteValueDictionary      Values { get; }
}
```

`RouteData` 具有 `Values` 和 `DataTokens` 两个属性，它们都是一个具有如下定义的类型为 `System.Web.Routing.RouteValueDictionary` 的对象。`RouteValueDictionary` 是一个用于保存路由变量的字典，其 `Key` 和 `Value` 分别表示变量的名称和值。`RouteData` 的 `Values` 属性的变量是路由对象通过对请求 URL 的解析得到的，而 `DataTokens` 属性则是直接附加到路由对象上的自定义变量。

```
public class RouteValueDictionary :
    IDictionary<string, object>,
    ICollection<KeyValuePair<string, object>>,
    IEnumerable<KeyValuePair<string, object>>,
    IEnumerable
{
    //省略成员
}
```

在某些路由场景中，我们要求路由对象针对当前请求进行解析得到的变量集合（`Values` 属性）必须包含某些固定名称的变量值（比如 ASP.NET MVC 应用中表示 `Controller` 和 `Action` 名称的变量），而 `GetRequiredString` 方法用于获取指定名称的变量值。对于该方法的调用，如果指定名称的变量在 `Values` 属性中不存在，则直接抛出一个 `InvalidOperationException` 异常。

`RouteData` 还具有另一个名称为 `RouteHandler` 的属性，其类型为具有如下定义的 `System.Web.Routing.IRouteHandler` 接口。`IRouteHandler` 接口在整个 URL 路由系统中具有重要的地位，其重要作用在于提供最终用于处理请求的 `HttpHandler` 对象（通过调用其 `GetHttpHandler` 方法获取）。我们可以在构造函数中对 `RouteData` 的 `RouteHandler` 属性进行初始化，也可以直接对该属性进行赋值。

```
public interface IHttpHandler
{
    IHttpHandler GetHttpHandler(RequestContext requestContext);
}
```

当请求被成功路由到某个.aspx 页面后，通过匹配路由对象的 `GetRouteData` 方法得到的 `RouteData` 对象被直接附加到目标页面对应的 `Page` 对象上。如下面的代码片段所示，`Page` 类型具有一个类型为 `RouteData` 的同名只读属性，返回的正是这个 `RouteData` 对象。

```
public class Page : TemplateControl, IHttpHandler
{
    //其他成员
    public RouteData RouteData { get; }
}
```

介绍完了作为 `RouteBase` 的 `GetRouteData` 方法的返回类型 `RouteData` 之后，我们接着介绍作为 `GetVirtualPath` 方法返回类型的 `VirtualPathData`。当 `RouteBase` 的 `GetVirtualPath` 方法被执行的时候，如果定义在 URL 模板中的变量与指定变量列表相匹配，则将指定的路由变量值替换 URL 模板中的变量占位符以生成一个虚拟路径。生成的虚拟路径与路由对象最终被封装成一个 `VirtualPathData` 对象返回，它们对应着 `VirtualPathData` 对象的 `VirtualPath` 和 `Route` 属性。另一个 `DataTokens` 属性和 `RouteData` 的同名属性一样都是来源于附加到路由对象的自定义变量集合。

```
public class VirtualPathData
{
    public VirtualPathData(RouteBase route, string virtualPath);

    public RouteValueDictionary DataTokens { get; }
    public RouteBase Route { get; set; }
    public string VirtualPath { get; set; }
}
```

`RouteBase` 的 `GetVirtualPath` 方法还涉及另一个 `System.Web.Routing.RequestContext` 类型。`RequestContext` 在 URL 路由系统和 ASP.NET MVC 路由体系中是一个频繁使用的类型，用于表示当前的请求上下文。从如下的代码片段中不难看出它实际上是对 HTTP 上下文和 `RouteData` 的封装。

```
public class RequestContext
{
    public RequestContext();
    public RequestContext(HttpContextBase httpContext, RouteData routeData);

    public virtual HttpContextBase HttpContext { get; set; }
    public virtual RouteData RouteData { get; set; }
}
```

Route

`System.Web.Routing.Route` 是抽象类 `RouteBase` 唯一的直接子类，基于 URL 模板模式的路由匹配规则就定义在 `Route` 中，在默认的情况下通过调用 `RouteCollection` 的 `MapPageRoute`

方法在全局路由表中添加的就是这么一个对象。如下面的代码片段所示，Route 具有一个字符串类型的属性 Url，它代表绑定在该路由对象的 URL 模板。

```
public class Route : RouteBase
{
    public Route(string url, IRouteHandler routeHandler);
    public Route(string url, RouteValueDictionary defaults,
        IRouteHandler routeHandler);
    public Route(string url, RouteValueDictionary defaults,
        RouteValueDictionary constraints, IRouteHandler routeHandler);
    public Route(string url, RouteValueDictionary defaults,
        RouteValueDictionary constraints,
        RouteValueDictionary dataTokens, IRouteHandler routeHandler);

    public override RouteData GetRouteData(HttpContextBase httpContext);
    public override VirtualPathData GetVirtualPath(RequestContext requestContext,
        RouteValueDictionary values);

    public RouteValueDictionary Constraints { get; set; }
    public RouteValueDictionary Defaults { get; set; }
    public RouteValueDictionary DataTokens { get; set; }
    public IRouteHandler RouteHandler { get; set; }
    public string Url { get; set; }
}
```

在默认的情况下，针对请求 URL 的路由通过 Route 对象来完成，而某个 Route 对象是否会被选择取决于请求的地址是否与对应的 URL 模板的模式相匹配。具体的匹配规则很简单，可以通过一个简单的例子来说明，假设我们具有如下一个模板表示获取某个地区（通过电话区号表示）未来 *N* 天的天气情况的 URL。

```
/weather/{areacode}/{days}
```

对于上述这个 URL 模板来说，通过分隔符“/”对其进行拆分得到 4 个基本的字符串，我们将它们称为“段 (Segment)”。对于组成某个段的内容，又可以分为“变量 (Variable)”和“文本 (Literal)”，前者通过采用花括号 (“{ }”) 对变量名的封装来表示（比如表示电话区号的 “{areacode}” 和天数的 “{days}”），后者则代表单纯的文字（比如 “weather”）。值得一提的是 URL 路由体系在字符比较时对于大小写是不敏感的，因为 URL 本来就不区分大小写。

对于一个试图进行匹配的地址来说，匹配成功需要有两个基本的条件，即该地址包含的段的数量和 URL 模板相同，对应的文本段内容也一致。按照这个匹配规则，下面这个 URL 和上面我们定义的 URL 模板是相匹配的。

```
/weather/0512/2
```

除了用于表示 URL 模板的核心属性 Url 之外，Route 还具有额外一些属性。属性 Constraints 为定义在 URL 模板中的变量以正则表达式的形式设定一些限制条件，该属性类型为 RouteValueDictionary，其 Key 和 Value 分别表示变量名和作为限制的正则表达式。比如对于上面定义的这个 URL 模板来说，我们为两个变量指定相应的正则表达式使请求地址具有合法的区号和作为整数的未来天数。如果我们通过该属性为 Route 对象定义了基于某些变

量的正则表达式，匹配成功的先决条件除了上述两个之外，被验证的 URL 中对应的段还必须通过对应的正则表达式的验证。

Route 另一个类型为 `RouteValueDictionary` 的属性 `Defaults` 为变量定义默认一个默认值，提供默认值的变量不要求一定要定义在 URL 模板中。当 Route 对给定 URL 进行匹配判断的时候，如果 URL 只能匹配模板前面的部分，但是后边部分均为变量段并且具有对应的默认值，这种情况下依然被视为成功匹配。还是以前面给出的 URL 模板为例，如果我们将 `{areacode}` 和 `{days}` 这两个变量的默认值分别设置为“010”（北京）和“2”（未来两天），如下所示的 3 个 URL 都能和该 Route 成功匹配，并且它们是等效的。

```
/weather/010/2
/weather/010
/weather/
```

关于定义在 URL 模板中的变量，我们并不要求它作为整个段的内容，换句话说，一个段可以同时包含文本和变量。此外，我们可以采用“`{*<<variable>>}`”来匹配 URL 的最后部分（可以包含多个段），而匹配的内容最终作为对应变量的值，姑且称之为“通配变量”。

```
/ {filename}.{extension} / { *pathinfo }
```

对于如上的这个 URL 模板，第一个段中包含两部分内容，即表示文件名称和扩展名的变量 `{filename}` 和 `{extension}`，以及作为两者分隔符的文本内容“.”，后边紧跟一个通配文变量 `{ *pathinfo }`。这个 URL 模板与下面一个 URL 是可以成功匹配的，匹配后定义在 URL 模板中的三个变量（`{filename}`、`{extension}` 和 `{pathinfo}`）的值分别为“default”、“aspx”和“abc/123”。

```
/default.aspx/abc/123
```

Route 的 `DataTokens` 属性在之前已经有所提及，它用于存储一些额外变量，它们不会参与针对请求地址的匹配工作。对于调用 Route 的 `GetRouteData` 和 `GetVirtualPath` 方法分别得到的 `RouteData` 和 `VirtualPathData` 对象，它们的 `DataTokens` 属性所包含的数据都来源于此。

RouteTable

对于一个 Web 应用来说，针对所有页面对应的 URL 不可能采用相同的模式，与之匹配的路由对象自然也不可能是唯一的。一个 Web 应用通过 `RouteTable` 的静态只读属性 `Routes` 维护一个全局的路由表，如下面的代码片段所示，该属性的类型为 `System.Web.Routing.RouteCollection`。

```
public class RouteTable
{
    public static RouteCollection Routes { get; }
}
```

顾名思义，`RouteCollection` 就是一个路由对象的集合，它提供了如下一些方法和属性。定义在 `RouteCollection` 中的方法和属性是为最终的开发人员设计的，我们针对 URL 路由系

统的编程所涉及的方法主要集中在这一类。

```
public class RouteCollection : Collection<RouteBase>
{
    //其他成员
    public RouteData GetRouteData(HttpContextBase httpContext);
    public VirtualPathData GetVirtualPath(RequestContext requestContext,
        RouteValueDictionary values);
    public VirtualPathData GetVirtualPath(RequestContext requestContext,
        string name, RouteValueDictionary values);

    public void Ignore(string url);
    public void Ignore(string url, object constraints);

    public Route MapPageRoute(string routeName, string routeUrl,
        string physicalFile);
    public Route MapPageRoute(string routeName, string routeUrl,
        string physicalFile, bool checkPhysicalUrlAccess);
    public Route MapPageRoute(string routeName, string routeUrl,
        string physicalFile, bool checkPhysicalUrlAccess,
        RouteValueDictionary defaults);
    public Route MapPageRoute(string routeName, string routeUrl,
        string physicalFile, bool checkPhysicalUrlAccess,
        RouteValueDictionary defaults, RouteValueDictionary constraints);
    public Route MapPageRoute(string routeName, string routeUrl,
        string physicalFile, bool checkPhysicalUrlAccess,
        RouteValueDictionary defaults, RouteValueDictionary constraints,
        RouteValueDictionary dataTokens);

    public bool AppendTrailingSlash { get; set; }
    public bool LowercaseUrls { get; set; }
    public bool RouteExistingFiles { get; set; }
}
```

当我们调用 `RouteCollection` 的 `GetRouteData` 和 `GetVirtualPath` 方法的时候，其内部会遍历集合中的每一个路由对象，并传入给定的参数调用同名方法直到找到一个与指定的请求 URL 相匹配的路由对象（返回值不为 `Null`），并返回相应的 `RouteData` 和 `VirtualPathData` 对象。如果集合中的任何一个路由对象都不匹配，则最终的返回结果是 `Null`。

`RouteCollection` 的 `RouteExistingFiles` 属性用于控制是否需要存在物理文件实施路由，也就是说如果请求的 URL 与某个物理文件的路径一致的情况下是否还需要对其实施路由。该属性默认值为 `False`，即注册的路由不会影响针对物理文件的请求。

`AppendTrailingSlash` 和 `LowercaseUrls` 这两个布尔类型的属性与方法 `GetVirtualPath` 有关，它们决定了对 URL 的正常化（Normalization）。具体来说，`AppendTrailingSlash` 表示是否需要在末尾添加“/”（如果没有），而 `LowercaseUrls` 则意味着是否需要将生成的 URL 转变成小写。

其实我们使用得较为频繁的还是 `MapPageRoute` 和 `Ignore` 方法，前者用于注册某个物理文件（路径）与 URL 模板之间的映射，其本质就是在本集合中添加一个 `Route` 对象。后者则与此相反，用于注册一个 URL 模板使路由系统可以忽略掉某些 URL。

2.1.4 路由映射

总的来说，我们可以通过 `RouteTable` 的静态属性 `Routes` 得到一个基于整个应用的全局路由表，通过上面的介绍我们知道这是一个类型为 `RouteCollection` 的集合对象，可以通过调用它的 `MapPageRoute` 进行路由映射，即注册 URL 模板与某个物理文件的匹配关系。路由注册的核心就是在全局路由表中添加一个 `Route` 对象，该对象的绝大部分属性都可以通过 `MapPageRoute` 方法的相关参数来指定。接下来我们通过实现演示的方式来说明路由注册的一些细节问题。

前面给出了一个获取天气预报信息的 URL 模板，现在在 ASP.NET Web 应用中创建一个 `Weather.aspx` 页面，不过我们并不打算在该页面中呈现任何天气信息，而是将基于该页面的路由信息打印出来。该页面主体部分的 HTML 如下所示，不仅将基于当前页面的 `RouteData` 对象的 `Route` 和 `RouteHandler` 属性类型输出来，还将存储于 `Values` 和 `DataTokens` 字典的变量显示出来。

```
<form id="form1" runat="server">
  <div>
    <table>
      <tr>
        <td>Route:</td>
        <td><%=RouteData.Route != null?
          RouteData.Route.GetType().FullName:"" %></td>
      </tr>
      <tr>
        <td>RouteHandler:</td>
        <td><%=RouteData.RouteHandler != null?
          RouteData.RouteHandler.GetType().FullName:"" %></td>
      </tr>
      <tr>
        <td>Values:</td>
        <td>
          <ul>
            <%foreach (var variable in RouteData.Values)
              {%>
            <li>
              <%=variable.Key%>=<%=variable.Value%></li>
            <% }%>
          </ul>
        </td>
      </tr>
      <tr>
        <td>DataTokens:</td>
        <td>
          <ul>
            <%foreach (var variable in RouteData.DataTokens)
              {%>
            <li>
              <%=variable.Key%>=<%=variable.Value%></li>
            <% }%>
          </ul>
        </td>
      </tr>
    </table>
  </div>
</form>
```

在添加的 `Global.asax` 文件中,我们将路由注册操作定义在 `Application_Start` 方法中,如下面的代码片段所示,映射到 `weather.aspx` 页面的 URL 模板为 “`{areacode}/{days}`”。在调用 `MapPageRoute` 方法的时候,我们还为定义在 URL 模板的两个变量定义了默认值以及正则表达式。除此之外,我们还在注册的路由对象上附加了两个变量,表示对变量默认值的说明 (`defaultCity: BeiJing; defaultDays: 2`)。顺便说一下, `MapPageRoute` 方法中布尔类型的参数 `checkPhysicalUrlAccess` 表示是否需要表示被路由的目标地址的 URL 实施授权(针对原请求地址的 URL 授权总是会执行)。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        var defaults = new RouteValueDictionary {
            { "areacode", "010" }, { "days", 2 } };
        var constraints = new RouteValueDictionary {
            { "areacode", @"0\d{2,3}" }, { "days", @"[1-3]{1}" } };
        var dataTokens = new RouteValueDictionary {
            { "defaultCity", "BeiJing" }, { "defaultDays", 2 } };
        RouteTable.Routes.MapPageRoute("default", "{areacode}/{days}",
            "~/weather.aspx", false, defaults, constraints, dataTokens);
    }
}
```

变量默认值

由于我们为定义在 URL 模板中表示区号和天数的变量定义了默认值 (`areacode: 010; days: 2`),如果希望返回北京地区未来两天的天气,可以直接访问应用根地址,也可以只指定具体区号,或者同时指定区号和天数。如图 2-2 所示,当我们在浏览器地址栏中输入上述三种不同的 URL 会得到相同的输出结果。

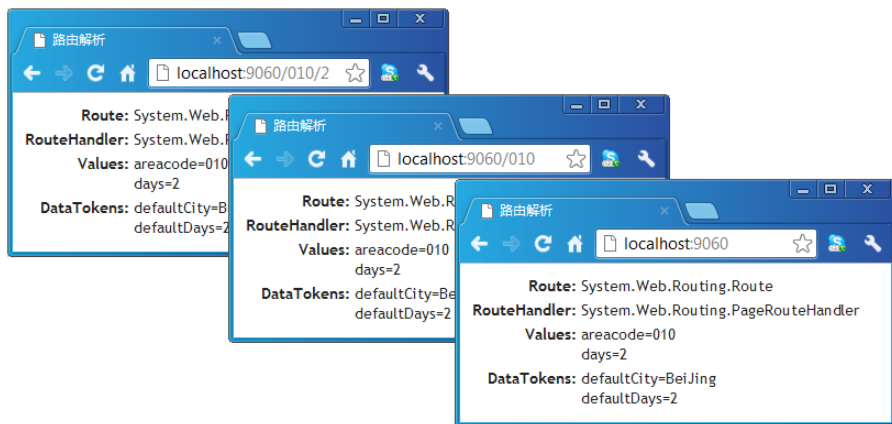


图 2-2 基于变量默认值的 URL 等效性

从图 2-2 所示的路由信息中可以看到,默认情况下 `RouteData` 的 `Route` 属性类型正是

`System.Web.Routing.Route`, 而 `RouteHandler` 属性则是一个 `System.Web.Routing.PageRouteHandler` 对象, 我们会在本章后续部分对 `PageRouteHandler` 进行详细介绍。通过地址解析出来的变量被保存在 `Values` 属性中, 而在进行路由注册过程为 `Route` 对象的 `DataTokens` 属性指定的变量被转移到了 `RouteData` 的同名属性中。(S202)

约束

我们以电话区号代表对应的城市, 为了确保用户在请求地址中提供有效的区号, 通过正则表达式 (“`0\d{2,3}`”) 对其进行了约束。此外, 假设只能提供未来 3 天以内的天气情况, 我们同样通过正则表达式 (“`[1-3]{1}`”) 对请求地址中表示天数的变量进行了约束。如果请求地址中的内容不能符合相关变量段的约束条件, 则意味着对应的路由对象与之不匹配。

对于本例来说, 由于只注册了唯一的路由对象, 如果请求地址不能满足我们定义的约束条件, 则意味着找不到一个具体目标文件, 会返回 404 错误, 如图 2-3 所示, 由于在请求地址中指定了不合法的区号 (01) 和天数 (4), 我们直接在浏览器界面上得到一个 HTTP 404 错误。(S202)

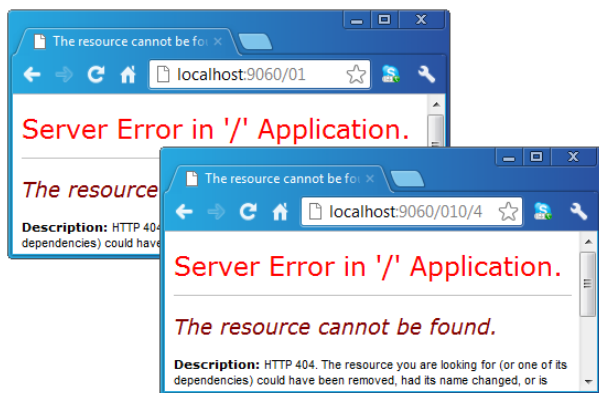


图 2-3 不满足正则表达式约束导致的 404 错误

对于约束, 除了可以通过字符串的形式为某个变量定义相应的正则表达式之外, 还可以指定一个实现了 `System.Web.Routing.IRouteConstraint` 接口类型的对象对整个请求进行约束。如下面的代码片段所示, `IRouteConstraint` 具有唯一的方法 `Match` 用于定义约束的逻辑, 该方法的 5 个参数分别表示: HTTP 上下文、当前路由对象、约束的名称 (存储约束对象在 `RouteValueDictionary` 中对应的 Key)、解析被匹配 URL 得到的变量集合以及表示路由的方向。

```
public interface IRouteConstraint
{
    bool Match(HttpContextBase httpContext, Route route, string parameterName,
        RouteValueDictionary values, RouteDirection routeDirection);
}
```

```
public enum RouteDirection
{
    IncomingRequest,
    UrlGeneration
}
```

所谓路由的方向表示是针对请求匹配（入栈）还是针对 URL 的生成（出栈），分别通过如上所示的枚举类型 `System.Web.Routing.RouteDirection` 的两个枚举值表示。具体来说，当调用路由对象的 `GetRouteData` 和 `GetVirtualPathData` 方法时，分别采用枚举值 `IncomingRequest` 和 `UrlGeneration`。

ASP.NET 路由系统的应用编程接口中定义了如下一个实现了 `IRouteConstraint` 接口的 `HttpMethodConstraint` 类型，顾名思义，`HttpMethodConstraint` 提供针对 HTTP 方法（GET、POST、PUT、DELETE 等）的约束，可以通过 `HttpMethodConstraint` 为路由对象设置一个允许的 HTTP 方法列表，只有在这个指定的列表中的 HTTP 方法名称的 HTTP 请求才允许被路由。这个被允许路由的 HTTP 方法列表对应于 `HttpMethodConstraint` 的只读属性 `AllowedMethods`，并在构造函数中初始化。

```
public class HttpMethodConstraint : IRouteConstraint
{
    public HttpMethodConstraint(params string[] allowedMethods);

    bool IRouteConstraint.Match(HttpContextBase httpContext, Route route,
        string parameterName, RouteValueDictionary values,
        RouteDirection routeDirection);

    public ICollection<string> AllowedMethods { get; }
}
```

同样是针对前面演示的例子，我们在进行路由注册的时候通过如下的代码应用了一个类型为 `HttpMethodConstraint` 的约束，并将允许的 HTTP 方法设置为 POST，意味着被注册的 `Route` 对象仅限于路由 POST 请求。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        var defaults = new RouteValueDictionary { { "areacode", "010" },
            { "days", 2 } };
        var constraints = new RouteValueDictionary { { "areacode", @"0\d{2,3}" },
            { "days", @"[1-3]{1}" },
            { "httpMethod", new HttpMethodConstraint("POST") } };
        var dataTokens = new RouteValueDictionary { { "defaultCity", "BeiJing" },
            { "defaultDays", 2 } };
        RouteTable.Routes.MapPageRoute("default", "{areacode}/{days}",
            "~/weather.aspx", false, defaults, constraints, dataTokens);
    }
}
```

现在我们采用与注册的 URL 模板相匹配的地址（/010/2）来访问 `Weather.aspx` 页面，依然会得到如图 2-4 所示的 HTTP 404 错误。（S203）

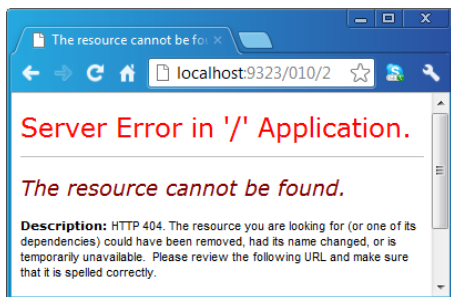


图 2-4 不满足 HTTP 方法约束（POST）导致的 404 错误

对现有物理文件的路由

在成功注册路由的情况下，如果我们按照传统的方式访问一个物理文件（比如.aspx、.css 或者.js 等），在请求地址满足某个路由的 URL 模板模式的情况下，ASP.NET 是否还是正常实施路由呢？不妨通过实例来测试一下。为了让针对某个物理文件的访问地址也满足注册路由对象的 URL 模板模式，我们需要按照如下的方式在进行路由注册时将表示约束的参数设置为 Null。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        var defaults = new RouteValueDictionary {
            { "areacode", "010" }, { "days", 2 } };
        var dataTokens = new RouteValueDictionary {
            { "defaultCity", "BeiJing" }, { "defaultDays", 2 } };
        RouteTable.Routes.MapPageRoute("default", "{areacode}/{days}",
            "~/weather.aspx", false, defaults, null, dataTokens);
    }
}
```

当通过传统的方式来访问存放于根目录下的 weather.aspx 页面时会得到如图 2-5 所示的结果，从界面上的输出结果不难看出，虽然请求地址完全满足我们注册路由对象的 URL 模板模式，但是 ASP.NET 并没有对请求地址实施路由。原因很简单，如果中间发生了路由，基于页面的 RouteData 的各项属性都不可能为空。（S204）

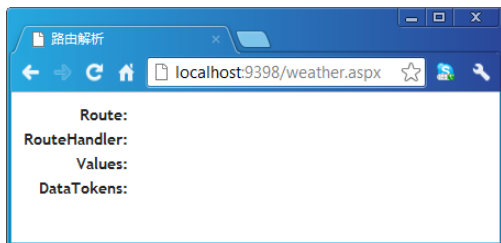


图 2-5 直接请求现存的物理文件（RouteExistingFiles = false）

如果请求地址对应着一个现存的物理文件,ASP.NET 会不会总是自动忽略路由呢? 实则不然,不对现有文件实施路由仅仅是默认采用的行为而已,是否对现有文件实施路由取决于代表全局路由表的 `RouteCollection` 对象的 `RouteExistingFiles` 属性,该属性默认情况下为 `False`,可以将此属性设置为 `True` 使 ASP.NET 路由系统忽略现有物理文件的存在,总是按照注册的路由表进行路由。为了演示这种情况,我们对 `Global.asax` 文件作了如下改动,在进行路由注册之前将 `RouteTable` 的 `Routes` 属性代表的 `RouteCollection` 对象的 `RouteExistingFiles` 属性设置为 `True`。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.RouteExistingFiles = true;
        var defaults = new RouteValueDictionary {
            { "areacode", "010" }, { "days", 2 } };
        var dataTokens = new RouteValueDictionary {
            { "defaultCity", "BeiJing" }, { "defaultDays", 2 } };
        RouteTable.Routes.MapPageRoute("default", "{areacode}/{days}",
            "~/weather.aspx", false, defaults, null, dataTokens);
    }
}
```

依旧是针对 `weather.aspx` 页面的访问,却得到了不一样的结果。从图 2-6 中可以看到,针对页面的相对地址 `weather.aspx` 不再指向具体的 Web 页面,在这里就是一个表示获取的天气信息对应的目标城市 (`areacode=weather.aspx`)。(S205)

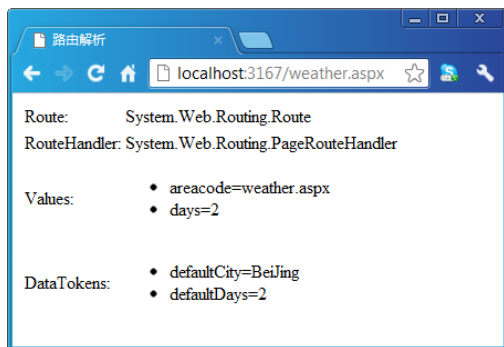


图 2-6 直接请求现存的物理文件 (`RouteExistingFiles = true`)

注册路由忽略地址

如果将代表全局路由表的 `RouteTable` 的静态属性 `Routes` 的 `RouteExistingFiles` 属性设置为 `True`,意味着 ASP.NET 针对所有抵达的请求都按照注册的路由表进行注册,但这同样会带来一些问题。不知道读者有没有发现图 2-5 和图 2-2/2-4 所示的页面具有不一样的样式,这是因为我们可以在页面中按照如下的方式引用一个 `.css` 文件来定义样式。

```
<link rel="stylesheet" href="Style.css" />
```

由于我们将全局路由表的 `RouteExistingFiles` 属性设置为 `True`，意味着针对上面这个 `Style.css` 文件的访问也会被路由。根据我们注册的路由规则，针对这个文件的访问会被自动导向 `weather.aspx` 这个页面，JS 脚本文件名被当成了路由变量 `{areaCode}` 的值。如图 2-7 所示，我们直接在浏览器的地址栏中输入 `Style.css` 文件的地址，呈现出来还是我们所熟悉的界面（`areacode= Style.css`）。（S205）¹

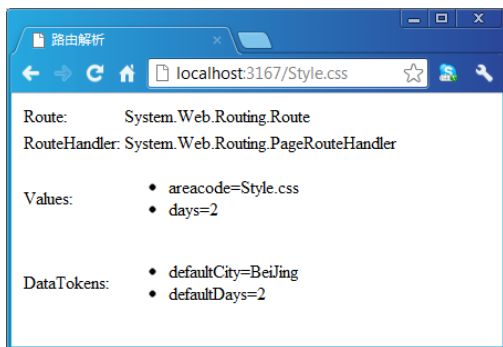


图 2-7 直接请求现存的.js 文件（`RouteExistingFiles = true`）

这是一个不得不解决的问题，因为它使我们无法正常地在页面中引用 JavaScript 和 CSS 文件，可以通过调用 `RouteCollection` 的 `Ignore` 方法来注册一些需要让路由系统忽略的 URL 模板。从前面给出的关于 `RouteCollection` 的定义中可以看到它具有两个 `Ignore` 方法重载，除了指定需要忽略的 URL 模板之外，还可以对相关的变量定义约束正则表达式。为了让路由系统忽略掉针对 CSS 文件请求，我们可以按照如下的方式在 `Global.asax` 中调用 `RouteTable` 的 `Routes` 属性的 `Ignore` 方法。值得一提的是这样的方法调用应该放在路由注册之前，否则起不到任何作用。（S206）

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.RouteExistingFiles = true;
        RouteTable.Routes.Ignore("{filename}.css/{*pathInfo}");
        //其他操作
    }
}
```

直接添加路由对象

我们调用 `RouteCollection` 对象的 `MapPageRoute` 方法进行路由注册的本质就在路由字典

¹ 由于 URL 路由是实现在 ASP.NET 中，在采用 IIS 作为宿主的情况下，对于 IIS 6.0 之前的版本或者 Classic 模式下的 IIS 7.0，针对静态.css 文件的请求根本不会进入 ASP.NET 管道，所以对.css 文件的请求是会被路由的。我们这个例子是寄宿在 Visual Studio Development Server 中。

中添加 `Route` 对象，所以我们完全可以调用 `Add` 方法添加一个手工创建的 `Route` 对象，如下所示的两种路由注册方式是完全等效的。如果需要添加一个继承自 `RouteBase` 的自定义路由对象，我们不得不采用手工添加的方式。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        var defaults = new RouteValueDictionary {
            { "areacode", "010" }, { "days", 2 } };
        var constraints = new RouteValueDictionary {
            { "areacode", @"0\d{2,3}" }, { "days", @"[1-3]{1}" } };
        var dataTokens = new RouteValueDictionary {
            { "defaultCity", "BeiJing" }, { "defaultDays", 2 } };

        //路由注册方式 1
        RouteTable.Routes.MapPageRoute("default", "{areacode}/{days}",
            "~/weather.aspx", false, defaults, constraints, dataTokens);

        //路由注册方式 2
        Route route = new Route("{areacode}/{days}", defaults, constraints,
            dataTokens, new PageRouteHandler("~/weather.aspx", false));
        RouteTable.Routes.Add("default", route);
    }
}
```

2.1.5 根据路由规则生成 URL

前面我们已经提到过 ASP.NET 的路由系统主要有两个方面的应用，一个就是通过注册 URL 模板与物理文件路径的匹配实现请求地址和物理地址的分离，另一个则是通过注册的路由规则生成一个相应的 URL，后者通过调用 `RouteCollection` 对象的 `GetVirtualPath` 方法来实现。

如下面的代码片段所示，`GetVirtualPath` 定义了两个 `GetVirtualPath` 方法重载，它们共同的参数 `requestContext` 和 `values` 分别表示请求上下文（`RouteData` 和 HTTP 上下文的封装）和用于替换定义在 URL 模板中的变量占位符的值。另一个 `GetVirtualPath` 方法具有一个额外的字符串参数 `name`，它表示集合中具体使用的路由对象的注册名称（调用 `MapPageRoute` 方法时指定的第一个参数）。

```
public class RouteCollection : Collection<RouteBase>
{
    //其他成员
    public VirtualPathData GetVirtualPath(RequestContext requestContext,
        RouteValueDictionary values);
    public VirtualPathData GetVirtualPath(RequestContext requestContext,
        string name, RouteValueDictionary values);
}
```

如果调用 `GetVirtualPath` 方法时没有指定具体生成虚拟路径的路由对象，该方法会遍历

整个集合直到找到一个 URL 模板与指定的路由参数列表相匹配的路由对象，并返回它生成的封装虚拟路径的 `VirtualPathData` 对象。具体来说就是依次调用集合中每个路由对象的 `GetVirtualPath` 方法，直到方法返回的 `VirtualPathData` 对象不为 `Null`，而该 `VirtualPathData` 对象则作为整个方法调用的返回值。如果所有路由的 `GetVirtualPath` 方法返回值均为 `Null`，那么整个方法的返回值也为 `Null`。

我们在调用 `GetVirtualPath` 方法的时候可以传入 `Null` 作为第一个参数(`requestContext`)，在这种情况下会基于当前 HTTP 上下文（对应于 `HttpContext` 的静态属性 `Current`）创建一个 `RequestContext` 对象作为调用路由对象 `GetVirtualPath` 方法的参数，该参数包含一个空的 `RouteData` 对象。如果当前 HTTP 上下文不存在则直接抛出一个 `InvalidOperationException` 异常。

路由对象针对 `GetVirtualPath` 方法而进行的路由匹配只要求 URL 模板中定义的变量的值都能被提供，而这些变量值具有三种来源，分别是路由对象定义的默认变量值、指定 `RequestContext` 的 `RouteData` 提供的变量值（`Values` 属性）和手工提供的变量值（通过 `values` 参数指定的 `RouteValueDictionary` 对象），这三种变量值具有由低到高的选择优先级。

同样以之前定义的关于获取天气信息的 URL 模板为例，我们在 `Weather.aspx` 页面的后台代码中通过如下的代码调用 `RouteTable` 和 `Routes` 属性的 `GetVirtualPath` 方法将生成三个具体的 URL。

```
public partial class Weather : Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        RouteData routeData = new RouteData();
        routeData.Values.Add("areaCode", "0512");
        routeData.Values.Add("days", "1");
        RequestContext requestContext = new RequestContext();
        requestContext.HttpContext = new HttpContextWrapper(HttpContext.Current);
        requestContext.RouteData = routeData;

        RouteValueDictionary values = new RouteValueDictionary();
        values.Add("areaCode", "028");
        values.Add("days", "3");

        Response.Write(RouteTable.Routes.GetVirtualPath(null, null).VirtualPath
            + "<br/>");
        Response.Write(RouteTable.Routes.GetVirtualPath(requestContext,
            null).VirtualPath + "<br/>");
        Response.Write(RouteTable.Routes.GetVirtualPath(requestContext,
            values).VirtualPath + "<br/>");
    }
}
```

从上面的代码片段可以看到，第一次调用 `GetVirtualPath` 方法传输的 `requestContext` 和 `values` 参数均为 `Null`；第二次则指定了一个手工创建的 `RequestContext` 对象，其 `RouteData` 的 `Values` 属性具有两个变量（`areaCode=0512`；`days=1`），而 `values` 参数依然为 `Null`；第三次

我们同时为参数 `requestContext` 和 `values` 指定了具体的对象，而后者包含两个参数（`areaCode=028`；`days=3`）。在浏览器上访问 `Weather.aspx` 页面会得到如图 2-8 所示的 3 个 URL，这充分证实了上面提到的关于变量选择优先级的结论。（S207）

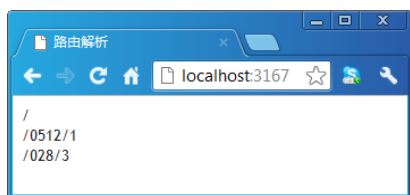


图 2-8 GetVirtualPath 方法接受不同参数生成的 URL

2.2 ASP.NET MVC 扩展

ASP.NET 的路由系统旨在通过注册 URL 模板与物理文件之间的映射进而实现请求地址与文件路径之间的分离，但是对于 ASP.NET MVC 应用来说，请求的目标不再是一个具体的物理文件，而是定义在某个 `Controller` 类型中的 `Action` 方法。出于自身路由特点的需要，ASP.NET MVC 对 ASP.NET 的路由系统进行了相应的扩展。

2.2.1 路由映射

通过前面的介绍我们知道，基于某个物理文件的路由注册通过调用 `RouteTable` 的静态属性 `Routes`（一个代表全局路由表的 `RouteCollection` 对象）的 `MapPageRoute` 方法来完成。为了实现针对目标 `Controller` 和 `Action` 的路由，ASP.NET MVC 针对 `RouteCollection` 类型定义了一系列的扩展方法以实现文件路径无关的路由映射，这些扩展方法定义在 `System.Web.Mvc.RouteCollectionExtensions` 类型中。

如下面的代码片段所示，`RouteCollectionExtensions` 定义了两组方法，方法 `IgnoreRoute` 用于注册不需要进行路由的 URL 模板，对应于 `RouteCollectionExtensions` 的 `Ignore` 方法；方法 `MapRoute` 用于进行基于 URL 模板的路由注册，对应于 `RouteCollectionExtensions` 的 `MapPageRoute` 方法。

```
public static class RouteCollectionExtensions
{
    //其他成员
    public static void IgnoreRoute(this RouteCollection routes, string url);
    public static void IgnoreRoute(this RouteCollection routes, string url,
        object constraints);

    public static Route MapRoute(this RouteCollection routes, string name,
        string url);
    public static Route MapRoute(this RouteCollection routes, string name,
        string url, object defaults);
}
```



```

public static Route MapRoute(this RouteCollection routes, string name,
    string url, string[] namespaces);
public static Route MapRoute(this RouteCollection routes, string name,
    string url, object defaults, object constraints);
public static Route MapRoute(this RouteCollection routes, string name,
    string url, object defaults, string[] namespaces);
public static Route MapRoute(this RouteCollection routes, string name,
    string url, object defaults, object constraints, string[] namespaces);
}

```

由于 ASP.NET MVC 的路由注册与具体的物理文件无关，所以 `MapRoute` 方法中并没有一个表示文件路径的 `physicalFile` 参数。与直接定义在 `RouteCollectionExtensions` 中的 `Ignore` 和 `MapPageRoute` 方法不同的是，表示默认变量的参数 `defaults` 和基于正则表达式的变量约束的参数 `constraints` 都不再是一个 `RouteValueDictionary` 对象，而是一个普通的 `object`。这主要是为了编程上的便利，可以使我们通过匿名类型对象的方式来指定这两个参数值。该方法在内部会通过反射的方式得到指定对象的属性列表，并转换为 `RouteValueDictionary` 对象，其属性名和属性值可以作为字典元素的 `Key` 和 `Value`。

对于 ASP.NET MVC 来说，URL 路由系统对请求地址进行解析后生成的路由数据中必须包含目标 `Controller` 的名称。由于 `Controller` 名称仅仅对应着类型的名称，但是激活 `Controller` 实例的前提是我们能够正确地解析出它的具体类型，所以在具有多个同名 `Controller` 类型时可能需要用到命名空间。在调用 `MapRoute` 方法的时候可以通过字符串数组类型的参数 `namespaces` 来指定一个命名空间的列表。对于注册的命名空间，可以指定一个代表完整命名空间的字符串，也可以使用 “*” 作为通配符表示对命名空间相应的部分不作任何约束。

添加的命名空间列表最终是被存储于 `Route` 对象的 `DataTokens` 属性中，对应的 `Key` 为 “Namespaces”。`MapRoute` 方法没有为初始化 `Route` 对象的 `DataTokens` 属性提供相应的参数，如果没有指定命名空间列表，所有通过该方法添加的 `Route` 对象的 `DataTokens` 属性总是一个空的 `RouteValueDictionary` 对象。

对于针对定义在某个 `Controller` 中的某个 `Action` 的请求，如果注册的路由表与之匹配，具体匹配的路由对象的 `GetRouteData` 方法被调用并返回一个具体的 `RouteData` 对象。对请求地址进行解析得到的目标 `Controller` 和 `Action` 的名称必须包含在该 `RouteData` 的 `Values` 属性对应的 `RouteValueDictionary` 对象中，其对应的 `Key` 分别为 `controller` 和 `action`。

2.2.2 实例演示：注册路由映射与查看路由信息（S208）

ASP.NET MVC 通过 `RouteCollection` 中的扩展方法 `MapRoute` 进行路由映射，为了让读者对此有一个深刻的认识，我们来进行一个简单的实例演示。依然沿用之前关于获取天气信息的场景，看看通过这种方式进行注册的 `Route` 对象针对匹配的 HTTP 请求将返回怎样的 `RouteData` 对象。

我们在创建的 ASP.NET Web 应用（不是 ASP.NET MVC 应用）中添加一个 Web 页面

(Default.aspx)，并按照之前的做法以内联代码的方式直接将 `RouteData` 的相关属性显示出来，页面主体部分的 HTML 如下所示。需要注意的是我们显示的 `RouteData` 是从定义的方法 `GetRouteData` 获取的，而不是对应于当前页面的 `RouteData` 属性。

```
<form id="form1" runat="server">
    <div>
        <table>
            <tr>
                <td>Route:</td>
                <td><%=GetRouteData().Route != null?
                    GetRouteData().Route.GetType().FullName:"" %></td>
            </tr>
            <tr>
                <td>RouteHandler:</td>
                <td><%=GetRouteData().RouteHandler != null?
                    GetRouteData().RouteHandler.GetType().FullName:"" %></td>
            </tr>
            <tr>
                <td>Values:</td>
                <td>
                    <ul>
                        <%=foreach (var variable in GetRouteData().Values)
                            {%>
                            <li>
                                <%=variable.Key%>=<%=variable.Value%></li>
                            <% }%>
                        </ul>
                    </td>
            </tr>
            <tr>
                <td>DataTokens:</td>
                <td>
                    <ul>
                        <%=foreach (var variable in GetRouteData().DataTokens)
                            {%>
                            <li>
                                <%=variable.Key%>=<%=variable.Value%></li>
                            <% }%>
                        </ul>
                    </td>
            </tr>
        </table>
    </div>
</form>
```

我们将 `GetRouteData` 方法定义在当前页面的后台代码中，如下面的代码片段所示，我们手工创建了一个 `HttpRequest` 和 `HttpResponse` 对象，`HttpRequest` 的请求的地址为“`http://localhost:3721/0512/3`”（3721 是本 Web 应用对应的端口号）。根据这两个对象创建了 `HttpContext` 对象，并以此创建一个 `HttpContextWrapper` 对象。最终将其作为参数调用 `RouteTable` 的 `Routes` 属性的 `GetRouteData` 方法并返回。这个方法实际上就是模拟注册的路由表针对相对地址为“`/0512/3`”的 HTTP 请求的路由处理。

```
public partial class Default : System.Web.UI.Page
{
```

```

private RouteData routeData;
public RouteData GetRouteData()
{
    if (null != routeData)
    {
        return routeData;
    }
    HttpRequest request = new HttpRequest("default.aspx",
        "http://localhost:3721/0512/3", null);
    HttpResponse response = new HttpResponse(new StringWriter());
    HttpContext context = new HttpContext(request, response);
    HttpContextBase contextWrapper = new HttpContextWrapper(context);
    return routeData = RouteTable.Routes.GetRouteData(contextWrapper);
}

```

具体的路由映射依然定义在添加的 `Global.asax` 文件中，如下面的代码片段所示，通过调用 `RouteTable` 的 `Routes` 属性的 `MapRoute` 方法注册了一个采用 “`{areacode}/{days}`” 作为 URL 模板的路由对象，并指定了默认变量、约束和命名空间列表。由于成功匹配的路由对象必须具有一个名为 “`controller`” 的路由变量，为了确保程序的成功运行，我们可以直接将 `controller` 的值设置为 “`Home`”。

```

public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        object defaults = new
        {
            areacode = "010",
            days = 2,
            defaultCity = "BeiJing",
            defaultDays = 2,
            controller = "Home"
        };
        object constraints = new { areacode = @"0\d{2,3}", days = @"[1-3]{1}" };
        string[] namespaces = new string[] {
            "Artech.Web.Mvc", "Artech.Web.Mvc.Html" };
        RouteTable.Routes.MapRoute("default", "{areacode}/{days}",
            defaults, constraints, namespaces);
    }
}

```

如果我们现在在浏览器中访问 `Default.aspx` 页面，会得到如图 2-9 所示的结果，从中可以得到一些有用的信息。

- 与调用 `RouteCollection` 的 `MapPageRoute` 方法进行路由映射不同的是，这个得到的 `RouteData` 对象的 `RouteHandler` 属性是一个 `System.Web.Mvc.MvcRouteHandler` 对象。
- 在 `MapRoute` 方法中通过 `defaults` 参数指定的两个与 URL 匹配无关的变量（`defaultCity=BeiJing`；`defaultDays=2`）体现在 `RouteData` 的 `Values` 属性中。这意味着如果我们没有在 URL 模板中为 `Controller` 和 `Action` 的名称定义相应的变量（`{controller}` 和 `{action}`），也可以将它们定义成默认变量。

- DataTokens 属性中包含一个 Key 为 “Namespaces” Value 为字符数组的元素，不难猜出它对应着我们指定的命名空间列表。

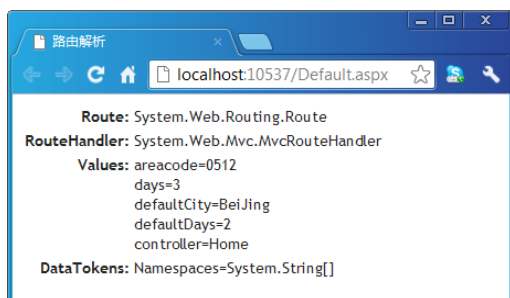


图 2-9 采用 ASP.NET MVC 路由映射得到的 RouteData

2.2.3 缺省 URL 参数

当通过 Visual Studio 的 ASP.NET MVC 项目模板创建一个 Web 应用后，它会为我们注册如下一个 URL 模板为“{controller}/{action}/{id}”的默认路由对象。三个路由变量(controller、action 和 id) 均具有相应的默认值，但是变量名为 id 的默认值为 `UrlParameter.Optional`。按照字面的意思，我们将其称为缺省 URL 参数，那么将默认值进行如此设置与设置一个具体的默认值有什么区别呢？

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        //其他操作
        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index",
                           id = UrlParameter.Optional }
        );
    }
}
```

在介绍缺省 URL 参数之前，不妨先来介绍定义它的 `System.Web.Mvc.UrlParameter` 类型的定义。如下面的代码片段所示，`UrlParameter` 是一个不能被实例化的类型（它具有唯一一个私有构造函数），唯一有用的就是用于定义缺省 URL 参数的静态只读字段 `Optional`。这是典型的单例编程模式，意味着多次注册的缺省 Url 参数引用着同一个 `UrlParameter` 对象。

```
public sealed class UrlParameter
{
    public static readonly UrlParameter Optional = new UrlParameter();

    private UrlParameter()
    {}
}
```

```

public override string ToString()
{
    return string.Empty;
}

```

在进行 URL 模式匹配的时候，默认值为缺省 Url 参数的路由变量与其他具有默认值的路由变量并没有什么差别。它们之间的不同之处在于如果将某个定义在 URL 模板中的变量的默认值定义为缺省 URL 参数，只有在请求 URL 中真正包含具体的变量值的情况下生成的 RouteData 的 Values 属性中才会包含相应的数据项。

举个简单的例子，我们在 ASP.NET MVC Web 应用²中直接使用如上所示的默认注册的路由。然后我们定义如下一个 HomeController，默认的 Action 方法 Index 具有一个名为 id 的参数，在该方法中将包含在当前 RouteData 对象的 Values 属性中的所有元素的 Key 和 Value 呈现出来。

```

public class HomeController : Controller
{
    public void Index(string id)
    {
        foreach (var variable in RouteData.Values)
        {
            Response.Write(string.Format("{0}: {1}<br/>",
                variable.Key, variable.Value));
        }
    }
}

```

我们直接运行该程序并在浏览器的地址栏中输入不同的 URL 来访问 HomeController 的 Action 方法 Index，看看最终包含在 RouteData 的路由变量有何不同。如图 2-10 所示，当直接通过根地址访问的时候，RouteData 的 Values 属性中只包含 controller 和 action 这两个变量，被设置为缺省 URL 参数的路由变量 id 只有在请求地址包含相应值的情况下才会出现在 RouteData 的 Values 属性中。（S209）

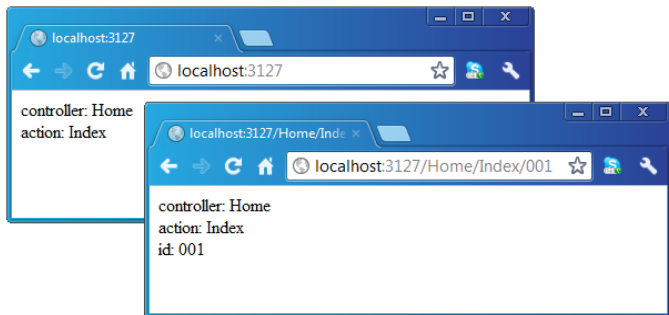


图 2-10 普通路由变量与缺省 URL 参数的路由变量之间的差别

2 本书用于实例演示而创建的 Web 应用，如果没有特殊说明就是通过 Visual Studio 的 ASP.NET MVC 项目模板创建的空 Web 应用。在必要的时候我们会添加一些 CSS 样式，但是具体的样式设置不会出现在给出的代码中。

2.2.4 基于 Area 的路由映射

对于一个较大规模的 Web 应用，可以从功能上通过 Area 将其划分为较小的单元。每个 Area 相当于一个独立的子系统，具有一套包含 Models、Views 和 Controller 在内的目录结构和配置文件。一般来说，每个 Area 具有各自的路由规则（URL 模板上一般会体现 Area 的名称），而基于 Area 的路由映射通过 System.Web.Mvc.AreaRegistration 进行注册。

AreaRegistration 与 AreaRegistrationContext

基于 Area 的路由映射通过 AreaRegistration 进行注册，如下面的代码片段所示，AreaRegistration 是一个抽象类，抽象只读属性 AreaName 返回当前 Area 的名称，而抽象方法 RegisterArea 用于实现基于当前 Area 的路由注册。

```
public abstract class AreaRegistration
{
    public static void RegisterAllAreas();
    public static void RegisterAllAreas(object state);

    public abstract void RegisterArea(AreaRegistrationContext context);
    public abstract string AreaName { get; }
}
```

AreaRegistration 定义了两个抽象的 RegisterAllAreas 方法重载，参数 state 表示传递给具体 AreaRegistration 的数据。当 RegisterAllArea 方法执行的时候，当前 Web 应用所有直接或者间接被引用的程序集会被加载（如果尚未加载），然后从这些程序集中解析出所有继承自 AreaRegistration 的类型并通过反射创建相应的 AreaRegistration 对象。针对每个 AreaRegistration 对象，一个 System.Web.Mvc.AreaRegistrationContext 对象被创建出来并作为参数调用它们的 RegisterArea 方法。

如下面的代码片段所示，AreaRegistrationContext 的只读属性 AreaName 表示 Area 的名称，属性 Routes 是一个代表路由表的 RouteCollection 对象，而 State 是一个用户自定义对象，它们均通过构造函数进行初始化。具体来说，AreaRegistrationContext 对象是在调用 AreaRegistration 的静态方法 RegisterAllAreas 对所有 Area 进行注册时被创建的，其 AreaName 来源于当前 AreaRegistration 对象的同名属性，Routes 则对应着 RouteTable 的静态属性 Routes 所表示的全局路由表。调用 RegisterAllAreas 方法指定的参数 state 值将被作为调用 AreaRegistrationContext 构造函数的同名参数。

```
public class AreaRegistrationContext
{
    public AreaRegistrationContext(string areaName, RouteCollection routes);
    public AreaRegistrationContext(string areaName, RouteCollection routes,
        object state);

    public Route MapRoute(string name, string url);
    public Route MapRoute(string name, string url, object defaults);
    public Route MapRoute(string name, string url, string[] namespaces);
    public Route MapRoute(string name, string url, object defaults,
```

```

        object constraints);
public Route MapRoute(string name, string url, object defaults,
    string[] namespaces);
public Route MapRoute(string name, string url, object defaults,
    object constraints, string[] namespaces);

public string          AreaName { get; }
public RouteCollection Routes { get; }
public object          State { get; }
public ICollection<string> Namespaces { get; }
}

```

`AreaRegistrationContext` 的只读属性 `Namespaces` 表示一组优先匹配的命名空间（当多个同名的 `Controller` 类型定义在不同的命名空间中）。当针对某个具体 `AreaRegistration` 的 `AreaRegistrationContext` 被创建的时候，如果 `AreaRegistration` 类型具有命名空间，在这个命名空间基础上添加 “.” 后缀生成的字符串会被添加到 `Namespaces` 集合中。换言之，对于多个定义在不同命名空间中的同名 `Controller` 类型，会优先选择包含在当前 `AreaRegistration` 命名空间下的 `Controller`。

`AreaRegistrationContext` 定义了一系列的 `MapRoute` 用于进行路由映射注册，方法的使用以及参数的含义与 `RouteCollection` 类的同名扩展方法一致。在这里需要特别指出的是，如果 `MapRoute` 方法没有指定命名空间，则通过属性 `Namespaces` 表示的命名空间列表会被使用，反之，该属性中包含的命名空间会被直接忽略。

当我们通过 Visual Studio 的 ASP.NET MVC 项目模板创建一个 Web 应用的时候，在 `Global.asax` 文件中会生成如下通过调用 `AreaRegistration` 的静态方法 `RegisterAllAreas` 实现对所有 `Area` 的注册的代码，也就是说，针对所有 `Area` 的注册发生在应用启动的时候。

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
    }
}

```

AreaRegistration 的缓存

`Area` 的注册（主要是基于 `Area` 的路由映射注册）通过具体的 `AreaRegistration` 来实现，在应用启动的时候，会遍历通过调用 `BuildManager` 的静态方法 `GetReferencedAssemblies` 方法得到的程序集列表，并从中找到所有 `AreaRegistration` 类型。如果一个应用涉及太多的程序集，这个过程可能会耗费很多时间。为了提高性能，ASP.NET MVC 会对解析出来的所有 `AreaRegistration` 类型列表进行缓存。

注：`BuildManager` 的静态方法 `GetReferencedAssemblies` 返回必须引用的程序集列表，这包括包含 `Web.config` 文件的 `<system.web>/<compilation>/<assemblies>` 配置节中指定的用于编译 Web 应用所使用的程序集和从 `App_Code` 目录中的自定义代码生成的程序集以及其他顶级文件夹中的程序集。

ASP.NET MVC 对 `AreaRegistration` 类型列表的缓存是基于文件的，具体来说，当通过程序集加载和反射得到了所有的 `AreaRegistration` 类型列表后，会对其序列化并保存为一个 XML 物理文件中，这个名为 `MVC-AreaRegistrationTypeCache.xml` 的 XML 文件被存放在 ASP.NET 的临时目录下，具体的路径如下。

- `%Windir%\Microsoft.NET\Framework\v{version}\TemporaryASP.NET Files\{appname}\...\UserCache\`
- `%Windir%\Microsoft.NET\Framework\v{version}\Temporary ASP.NET Files\root\...\UserCache\`

其中第一个针对寄宿于 IIS 中的 Web 应用，后者针对直接通过 Visual Studio Developer Server 作为宿主的应用。

下面的 XML 片段体现了这个作为所有 `AreaRegistration` 类型缓存的 XML 文件的结构，从中我们可以看到所有的 `AreaRegistration` 类型名称，连同它所在的托管模块和程序集名称都被保存了下来。当调用 `AreaRegistration` 的静态方法 `RegisterAllAreas` 被调用之后，系统会试图加载该文件，如果该文件存在并且具有期望的结构，那么将不再通过程序集加载和反射来解析所有 `AreaRegistration` 的类型，而是直接对文件内容进行反序列化，从而得到所有 `AreaRegistration` 类型的列表。

```
<?xml version="1.0" encoding="utf-8"?>
<!--This file is automatically generated. Please do not modify the contents of
this file.-->
<typeCache lastModified="3/22/2012 2:58:47 PM"
  mvcVersionId="80365b23-7a1d-42b2-9e7d-cc6f5694c6d1">
  <assembly name="Artech.Admin, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null">
    <module versionId="07be22a1-781d-4ade-bd22-34b0850445ef">
      <type>Artech.Admin.AdminAreaRegistration</type>
    </module>
  </assembly>
  <assembly name="Artech.Portal, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null">
    <module versionId="7b0490d4-427e-43cb-8cb5-ac1292bd4976">
      <type>Artech.Portal.PortalAreaRegistration</type>
    </module>
  </assembly>
</typeCache>
```

实例演示：查看基于 Area 路由信息（S210）

通过 `AreaRegistration` 实现的针对 Area 的路由注册具有一些特殊的细节差异，我们可以通过实例演示的方式来说明。直接使用前面创建的演示实例（S208），并在项目中创建一个自定义的 `WeatherAreaRegistration` 类。如下面的代码片段所示，`WeatherAreaRegistration` 继承自抽象基类 `AreaRegistration`，表示 Area 名称的 `AreaName` 属性返回“`Weather`”。在实现路由注册的 `RegisterArea` 方法中调用 `AreaRegistrationContext` 对象的 `MapRoute` 方法便注册了一个 URL 模板为“`weather/{areacode}/{days}`”的路由对象，相应的默认变量值、约束也会被提供。

```

public class WeatherAreaRegistration : AreaRegistration
{
    public override string AreaName
    {
        get { return "Weather"; }
    }

    public override void RegisterArea(AreaRegistrationContext context)
    {
        object defaults = new
        {
            areacode    = "010",
            days        = 2,
            defaultCity = "BeiJing",
            defaultDays = 2
        };
        object constraints = new { areacode = @"0\d{2,3}", days = @"[1-3]{1}" };
        context.MapRoute("weatherDefault", "weather/{areacode}/{days}", defaults,
            constraints);
    }
}

```

我们可以在 Global.asax 的 Application_Start 方法中按照如下的方式调用 AreaRegistration 的静态方法 RegisterAllAreas 来实现对所有 Area 的注册。按照上面介绍的 Area 注册原理，RegisterAllAreas 方法的第一次调用会自动加载所有引用的程序集来获取所有的 AreaRegistration（当然会包括我们上面定义的 WeatherAreaRegistration），最后通过反射创建相应的对象并调用 RegisterArea 方法。

```

public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        AreaRegistration.RegisterAllAreas();
    }
}

```

在用于获取路由信息的 GetRouteData 方法中，我们对创建的 HttpRequest 对象略加修改，使请求地址符合通过 WeatherAreaRegistration 注册的路由规则（“/weather/0512/3”）。

```

public partial class Default : System.Web.UI.Page
{
    private RouteData routeData;
    public RouteData GetRouteData()
    {
        if (null != routeData)
        {
            return routeData;
        }
        HttpRequest request = new HttpRequest("default.aspx",
            "http://localhost:3721/weather/0512/3", null);
        HttpResponse response = new HttpResponse(new StringWriter());
        HttpContext context = new HttpContext(request, response);
        HttpContextBase contextWrapper = new HttpContextWrapper(context);
        return routeData = RouteTable.Routes.GetRouteData(contextWrapper);
    }
}

```

在浏览器中访问 `Default.aspx` 页面，我们会得到如图 2-11 所示的结果。通过 `AreaRegistration` 注册的路由对象得到的 `RouteData` 的不同之处主要反映在其 `DataTokens` 属性上。如图 2-11 所示，除了表示命名空间列表的元素，`DataTokens` 属性表示的 `RouteValueDictionary` 还具有两个额外的元素，其中一个 `Key` 为“`area`”的元素代表 `Area` 的名称，另一个 `Key` 为“`UseNamespaceFallback`”的元素具有一个布尔值表示是否需要使用后备的命名空间来解析 `Controller` 类型。

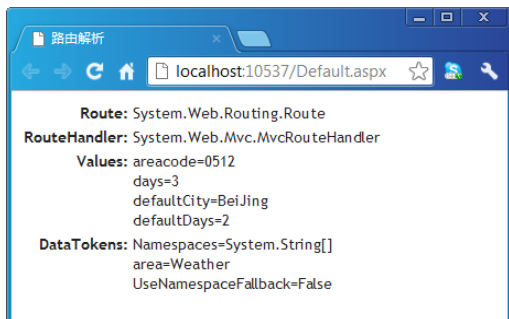


图 2-11 采用 `AreaRegistration` 路由映射得到的 `RouteData`

如果调用 `AreaRegistrationContext` 的 `MapRoute` 方法是显式指定了命名空间，或者说对应的 `AreaRegistration` 定义在某个命名空间下，这个名称为“`UseNamespaceFallback`”的 `DataToken` 元素的值为 `False`，反之为 `True`。进一步来说，如果在调用 `MapRoute` 方法时指定了命名空间列表，那么，`AreaRegistration` 类型所示在的命名空间会被忽略，也就是说，后者是前者的一个后备，前者具有更高的优先级。

`AreaRegistration` 类型所示在命名空间也不是直接作为最终 `RouteData` 的 `DataTokens` 中的命名空间，而是在此基础上加上“`*`”后缀。针对我们的实例来说，包含在 `RouteData` 的 `DataTokens` 集合中的命名空间为“`WebApp.*`”（`WebApp` 是定义 `WeatherAreaRegistration` 的命名空间）。

2.2.5 链接和 URL 的生成

ASP.NET 路由系统通过注册的路由表旨在实现两个“方向”的路由功能，即针对入栈请求的路由和出栈 URL 的生成。前者通过调用代表全局路由表的 `RouteCollection` 对象的 `GetRouteData` 方法实现，后者则依赖于 `RouteCollection` 的 `GetVirtualPathData` 方法，而最终还是落在继承自 `RouteBase` 的路由对象的同名方法的调用上。

ASP.NET MVC 定义了 `HtmlHelper` 和 `UrlHelper` 这两个帮助类，可以通过调用它们的 `ActionLink/RouteLink` 和 `Action/RouteUrl` 方法，并根据注册的路由规则生成相应的链接或者 URL。从本质上讲，`HtmlHelper/UrlHelper` 实现的对 URL 的生成最终还是依赖于前面所说的 `GetVirtualPathData` 方法。

UrlHelper V.S. HtmlHelper

在介绍如何通过 `HtmlHelper` 和 `UrlHelper` 来生成链接或者 URL 之前，先来看看它们的基本定义。从下面给出的代码片段我们可以看出，一个 `UrlHelper` 对象实际上对一个表示请求上下文的 `RequestContext` 对象和表示路由表的 `RouteCollection` 对象的封装，它们分别对应于只读属性 `RequestContext` 和 `RouteCollection`，并且在构造函数中被初始化。如果在构造 `UrlHelper` 的时候没有指定 `RouteCollection` 对象，那么通过 `RouteTable` 的静态属性 `Routes` 表示的全局路由表将直接被使用。

```
public class UrlHelper
{
    //其他成员
    public UrlHelper(RequestContext requestContext);
    public UrlHelper(RequestContext requestContext,
        RouteCollection routeCollection);

    public RequestContext      RequestContext { get; }
    public RouteCollection      RouteCollection { get; }
}
```

再来看看如下所示的 `HtmlHelper` 的定义，它同样具有一个表示路由对象集合的 `RouteCollection` 属性。和 `UrlHelper` 一样，如果在构造函数没有显式指定，`RouteTable` 的静态属性 `Routes` 表示的 `RouteCollection` 对象将会用于初始化该属性。

```
public class HtmlHelper
{
    //其他成员
    public HtmlHelper(ViewContext viewContext,
        IViewDataContainer viewDataContainer);
    public HtmlHelper(ViewContext viewContext,
        IViewDataContainer viewDataContainer, RouteCollection routeCollection);

    public RouteCollection      RouteCollection { get; }
    public ViewContext          ViewContext { get; }
}
public class ViewContext : ControllerContext
{
    //省略成员
}
public class ControllerContext
{
    //其他成员
    public RequestContext      RequestContext { get; set; }
    public virtual RouteData    RouteData { get; set; }
}
```

由于 `HtmlHelper` 只是在 `View` 中使用，所以它具有一个通过 `ViewContext` 属性表示的针对 `View` 的上下文。对于 `ViewContext`，我们会在第 8 章“`View` 的呈现”中对其进行单独介绍，在这里只需要知道它是表示 `Controller` 上下文的 `ControllerContext` 的子类，而后者通过 `RequestContext` 和 `RouteData` 属性获取当前的请求上下文和路由数据（其实 `RouteData` 属性表示的 `RouteData` 对象已经包含在 `RequestContext` 属性表示的 `RequestContext` 对象中）。

UrlHelper.Action V.S. HtmlHelper.ActionLink

UrlHelper 和 HtmlHelper 分别通过 Action 和 ActionLink 方法生成一个针对某个 Controller/Action 的 URL 和链接。下面的代码片段列出了 UrlHelper 的所有 Action 重载，参数 actionName 和 controllerName 分别代表 Action 和 Controller 的名称。通过 object 或者 RouteValueDictionary 类型表示的 routeValues 参数表示替换 URL 模板中变量的参数列表。参数 protocol 和 hostName 代表作为完整 URL 的传输协议（比如 http 和 https 等）以及主机名。

```
public class UrlHelper
{
    //其他成员
    public string Action(string actionName);
    public string Action(string actionName, object routeValues);
    public string Action(string actionName, string controllerName);
    public string Action(string actionName, RouteValueDictionary routeValues);
    public string Action(string actionName, string controllerName,
        object routeValues);
    public string Action(string actionName, string controllerName,
        RouteValueDictionary routeValues);

    public string Action(string actionName, string controllerName,
        object routeValues, string protocol);
    public string Action(string actionName, string controllerName,
        RouteValueDictionary routeValues, string protocol, string hostName);
}
```

对于定义在 UrlHelper 中的众多 Action 方法，如果我们显示指定了传输协议（protocol 参数）或者主机名称，返回的是一个完整的 URL，否则返回的是一个相对 URL。如果我们没有显示地指定 Controller 的名称（controllerName 参数），那么当前 Controller 的名称会被采用。对于 UrlHelper 来说，通过 RequestContext 属性表示的当前请求上下文包含了相应的路由信息（即 RequestContext 的 RouteData 属性表示的 RouteData）。RouteData 的 Values 属性中必须包含一个 Key 为“controller”的元素，其值就代表当前 Controller 的名称。

ASP.NET MVC 为 HtmlHelper 定义了如下所示的一系列 ActionLink 扩展方法重载，顾名思义，ActionLink 不再仅仅返回一个 URL，而是生成一个链接（<a>...），但是其中作为目标 URL 的生成逻辑与 UrlHelper 是完全一致的。

```
public static class LinkExtensions
{
    //其他成员
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, object routeValues);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, string controllerName);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, RouteValueDictionary routeValues);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, object routeValues,
```

```

        object htmlAttributes);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, RouteValueDictionary routeValues,
        IDictionary<string, object> htmlAttributes);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, string controllerName,
        object routeValues, object htmlAttributes);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, string controllerName,
        RouteValueDictionary routeValues,
        IDictionary<string, object> htmlAttributes);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, string controllerName,
        string protocol, string hostName, string fragment,
        object routeValues, object htmlAttributes);
    public static MvcHtmlString ActionLink(this HtmlHelper htmlHelper,
        string linkText, string actionName, string controllerName,
        string protocol, string hostName, string fragment,
        RouteValueDictionary routeValues,
        IDictionary<string, object> htmlAttributes);
}

```

实例演示：创建一个 RouteHelper 模拟 UrlHelper 的 URL 生成逻辑（S211）

为了让读者对 UrlHelper 如何利用 ASP.NET 路由系统进行 URL 生成的逻辑具有一个深刻认识，我们接下来创建一个名为 RouteHelper 的等效帮助类。如下面的代码片段所示，RouteHelper 具有 RequestContext 和 RouteCollection 两个属性，前者在构造函数中指定，后者直接返回通过 RouteTable 的 Routes 静态属性表示的全局路由表。

```

public class RouteHelper
{
    public RequestContext      RequestContext { get; private set; }
    public RouteCollection     RouteCollection { get; private set; }

    public RouteHelper(RequestContext requestContext)
    {
        this.RequestContext    = requestContext;
        this.RouteCollection    = RouteTable.Routes;
    }

    public string Action(string actionName, string controllerName=null,
        object routeValues=null, string protocol=null, string hostName = null)
    {
        controllerName = controllerName ??
            this.RequestContext.RouteData.GetRequiredString("controller");
        RouteValueDictionary routeValueDictionary =
            new RouteValueDictionary(routeValues);
        routeValueDictionary.Add("action", actionName);
        routeValueDictionary.Add("controller", controllerName);
        string virtualPath = this.RouteCollection.GetVirtualPath(
            this.RequestContext, routeValueDictionary).VirtualPath;
        if (string.IsNullOrEmpty(protocol) && string.IsNullOrEmpty(hostName))
        {
            return virtualPath.ToLower();
        }
    }
}

```

```

    }
    protocol = protocol ?? "http";
    Uri uri = this.RequestContext.HttpContext.Request.Url;
    hostName = hostName ?? uri.Host + ":" + uri.Port;
    return string.Format("{0}://{1}{2}", protocol,
        hostName, virtualPath).ToLower();
}
}

```

RouteHelper 定义了一个 Action 方法根据指定的 Action 名称、Controller 名称、路由参数列表、网络协议前缀和主机名称来生成相应的 URL，除了第一个表示 Action 的参数，其余参数均是可以缺省的。具体的逻辑很简单：如果指定的 Controller 名称为 Null，则通过 RequestContext 获取当前 Controller 名称，然后将 Action 和 Controller 名称添加到表示路由变参数的 RouteValueDictionary 对象中（routeValues 参数），对应的 Key 分别是“action”和“controller”。

然后调用 RouteCollection 的 GetVirtualPath 得到一个 VirtualPathData 对象，如果既没有显式指定传输协议也没有指定主机名称，直接返回 VirtualPathData 对象的 VirtualPath 属性体现的相对路径，否则生成一个完整的 URL。如果没有指定主机名称，我们采用当前请求的主机名称，并且使用当前的端口。如果没有指定传输协议，则直接使用“http”。

接下来在添加的 Global.asax 中通过如下的代码注册一个 URL 模板为“{controller}/{action}/{id}”的路由对象。

```

public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new
            {
                controller = "Home",
                action      = "Index",
                id          = UrlParameter.Optional
            }
        );
    }
}

```

在添加的 Web 页面（Default.aspx）中通过如下的代码利用我们自定义的 RouteHelper 生成 5 个 URL。在页面加载事件处理方法中，我们根据手工创建的 HttpRequest（请求地址为 http://localhost:3721/products/getproduct/001）和 HttpResponse 创建一个 HttpContext 对象，并进一步创建 HttpContextWrapper 对象。将此 HttpContextWrapper 对象作为参数调用全局路由表的 GetRouteData 方法得到封装路由数据的 RouteData 对象，并针对创建的 HttpContextWrapper 对象和此 RouteData 进一步创建 RequestContext 对象，最终创建出 RouteHelper 对象。

```

public partial class Default : System.Web.UI.Page
{

```

```
protected void Page_Load(object sender, EventArgs e)
{
    HttpRequest request = new HttpRequest("default.aspx",
        "http://localhost:3721/products/getproduct/001", null);
    HttpResponse response = new HttpResponse(new StringWriter());
    HttpContext context = new HttpContext(request, response);
    HttpContextBase contextWrapper = new HttpContextWrapper(context);

    RouteData routeData = RouteTable.Routes.GetRouteData(contextWrapper);
    RequestContext requestContext = new RequestContext(
        contextWrapper, routeData);
    RouteHelper helper = new RouteHelper(requestContext);

    Response.Write(helper.Action("GetProductCategories") + "<br/>");
    Response.Write(helper.Action("GetAllContacts", "Sales") + "<br/>");
    Response.Write(helper.Action("GetAllContact", "Sales",
        new { id = "001" }) + "<br/>");
    Response.Write(helper.Action("GetAllContact", "Sales",
        new { id = "001" }, "https") + "<br/>");
    Response.Write(helper.Action("GetAllContact", "Sales",
        new { id = "001" }, "https", "www.artech.com") + "<br/>");
}
}
```

运行该程序之后，通过调用 `RouteHelper` 的 `Action` 方法生成的 5 个 URL 会以图 2-12 所示的方式出现在浏览器上。

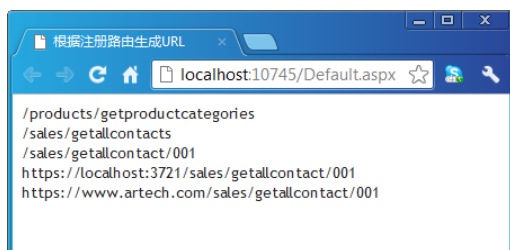


图 2-12 通过自定义 `RouteHelper` 生成的 URL

UrlHelper.RouteUrlV.S. HtmlHelper.RouteLink

不论是 `UrlHelper` 的 `Action` 方法，还是 `HtmlHelper` 的 `ActionLink`，URL 部分都是通过一个路由表生成出来的，而在默认的情况下这个路由表就是通过 `RouteTable` 的静态属性 `Routes` 表示的全局路由表。换句话说，具体使用的总是路由表中第一个匹配的路由对象，但是有时候我们需要针对注册的某个具体的路由对象来生成 URL 或者对应的链接，在这种情况下就需要使用到 `UrlHelper` 和 `HtmlHelper` 的另外一组方法了。

如下面的代码片段所示，`UrlHelper` 定义了一系列的 `RouteUrl` 方法，除了第一个重载之外，后面的重载都接受一个路由对象注册名称的参数 `routeName`。与 `UrlHelper` 的 `Action` 方法一样，可以指定用于替换定义在 URL 模板中路由变量的参数 (`routeValues`)，以及传输协议名称 (`protocol`) 和主机名称 (`hostName`)。

```

public class UrlHelper
{
    //其他成员
    public string RouteUrl(object routeValues);
    public string RouteUrl(string routeName);
    public string RouteUrl(RouteValueDictionary routeValues);
    public string RouteUrl(string routeName, object routeValues);
    public string RouteUrl(string routeName, RouteValueDictionary routeValues);
    public string RouteUrl(string routeName, object routeValues,
        string protocol);
    public string RouteUrl(string routeName, RouteValueDictionary routeValues,
        string protocol, string hostName);
}

```

对于没有指定路由对象注册名称的 `RouteUrl` 方法来说,它还是利用整个路由表进行 URL 的生成。如果显示指定了路由对象的注册名称,那么就会从路由表中获取相应的路由对象。如果该路由对象与指定的变量列表不匹配,则返回 `Null`, 否则返回生成的 URL。

`HtmlHelper` 也同样定义了类似的 `RouteLink` 方法重载用于实现基于指定路由对象的链接生成,具体的 `RouteLink` 方法定义如下。

```

public static class LinkExtensions
{
    //其他成员
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, object routeValues);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, RouteValueDictionary routeValues);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, object routeValues, object htmlAttributes);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName, object routeValues);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName, RouteValueDictionary routeValues);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, RouteValueDictionary routeValues,
        IDictionary<string, object> htmlAttributes);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName, object routeValues,
        object htmlAttributes);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName, RouteValueDictionary routeValues,
        IDictionary<string, object> htmlAttributes);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName, string protocol, string hostName,
        string fragment, object routeValues, object htmlAttributes);
    public static MvcHtmlString RouteLink(this HtmlHelper htmlHelper,
        string linkText, string routeName, string protocol, string hostName,
        string fragment, RouteValueDictionary routeValues,
        IDictionary<string, object> htmlAttributes);
}

```


2.3 动态 HttpHandler 映射

通过第1章“ASP.NET + MVC”对 ASP.NET 管道式设计的介绍，我们知道一个请求最终是通过一个具体的 `HttpHandler` 进行处理的。表示一个 Web 页面的 `Page` 对象就是一个 `HttpHandler`，它被用于最终处理基于某个 `.aspx` 文件的请求，可以通过 `HttpHandler` 的动态映射来实现请求地址与物理文件路径之间的分离。

实际上 ASP.NET 路由系统就是采用了这样的实现原理。如图 2-13 所示，ASP.NET 路由系统通过一个注册的自定义 `HttpModule` 实现对请求进行的拦截，然后动态映射一个用于处理当前请求的 `HttpHandler`。`HttpHandler` 对请求进行处理并最终对请求予以响应。

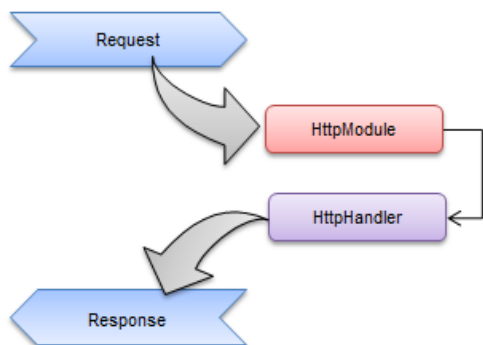


图 2-13 `HttpHandler` 的动态映射

2.3.1 `UrlRoutingModule`

图 2-13 所示的作为请求拦截器的 `HttpModule` 类型为 `System.Web.Routing.UrlRoutingModule`。如下面的代码片段所示，`UrlRoutingModule` 对请求的拦截是通过注册 `HttpApplication` 的 `PostResolveRequestCache` 事件实现的。

```
public class UrlRoutingModule : IHttpModule
{
    //其他成员
    public RouteCollection RouteCollection { get; set; }

    public void Init(HttpApplication context)
    {
        context.PostResolveRequestCache +=
            new EventHandler(this.OnApplicationPostResolveRequestCache);
    }

    private void OnApplicationPostResolveRequestCache(object sender,
        EventArgs e);
}
```

UrlRoutingModule 具有一个类型为 RouteCollection 的 RouteCollection 属性, 在默认的情况下该属性是对 RouteTable 的静态属性 Routes 的引用。用于最终处理请求的 HttpHandler 的动态映射就实现在 OnApplicationPostResolveRequestCache 方法中, 具体的实现逻辑非常简单: 通过 HttpApplication 获得当前的 HTTP 上下文, 并将其作为参数调用 RouteCollection 的 GetRouteData 方法得到一个 RouteData 对象。

通过 RouteData 的 RouteHandler 属性可以得到一个实现了 IRouteHandler 的 RouteHandler 对象, 调用后者的 GetHttpHandler 方法可以直接获取对应的 HttpHandler 对象, 而我们需要映射到当前请求的就是这么一个 HttpHandler。下面的代码片段基本上体现了定义在 UrlRoutingModule 的 OnApplicationPostResolveRequestCache 方法中的动态 HttpHandler 映射逻辑。

```
public class UrlRoutingModule : IHttpModule
{
    //其他成员
    private void OnApplicationPostResolveRequestCache(object sender, EventArgs e)
    {
        HttpContext context = ((HttpApplication)sender).Context;
        HttpContextBase contextWrapper = new HttpContextWrapper(context);
        RouteData routeData = this.RouteCollection.GetRouteData(contextWrapper);
        RequestContext requestContext =
            new RequestContext(contextWrapper, routeData);
        IHttpHandler handler =
            routeData.RouteHandler.GetHttpHandler(requestContext);
        context.RemapHandler(handler);
    }
}
```

2.3.2 PageRouteHandler 与 MvcRouteHandler

通过前面的介绍我们知道, 对于调用 RouteCollection 的 GetRouteData 获得的 RouteData 对象, 其 RouteHandler 来源于与当前请求 URL 相匹配的 Route 对象。对于通过调用 RouteCollection 的 MapPageRoute 方法注册的 Route 来说, 它的 RouteHandler 属性返回一个类型为 System.Web.Routing.PageRouteHandler 的对象。

由于调用 MapPageRoute 方法的目的在于实现请求地址与某个.aspx 页面文件之间的映射, 所以我们最终还是要创建 Page 对象来处理相应的请求, 所以 PageRouteHandler 的 GetHttpHandler 方法最终返回的就是针对映射页面文件路径的 Page 对象。此外, MapPageRoute 方法中还可以控制是否对物理文件地址实施授权, 而授权在返回 Page 对象之前进行。

定义在 PageRouteHandler 中的 HttpHandler 映射逻辑基本上体现在如下的代码片段中, 两个属性 VirtualPath 和 CheckPhysicalUrlAccess 表示页面文件的地址及是否需要物理文件地址实施 URL 授权, 它们在构造函数中被初始化, 且最初来源于调用 RouteCollection 的

MapPageRoute 方法传入的参数。

```
public class PageRouteHandler : IRouteHandler
{
    public bool        CheckPhysicalUrlAccess { get; private set; }
    public string      VirtualPath { get; private set; }

    public PageRouteHandler(string virtualPath, bool checkPhysicalUrlAccess)
    {
        this.VirtualPath = virtualPath;
        this.CheckPhysicalUrlAccess = checkPhysicalUrlAccess;
    }

    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        if (this.CheckPhysicalUrlAccess)
        {
            //Check Physical Url Access
        }
        return (IHttpHandler)BuildManager.CreateInstanceFromVirtualPath(
            this.VirtualPath, typeof(Page))
    }
}
```

ASP.NET MVC 的 Route 对象是通过调用 RouteCollection 的扩展方法 MapRoute 进行注册的，它对应的 RouteHandler 是一个类型为 System.Web.Mvc.MvcRouteHandler 的对象。如下面的代码片段所示，MvcRouteHandler 用于获取处理当前请求的 IHttpHandler 是一个 System.Web.Mvc.MvcHandler 对象。MvcHandler 实现对 Controller 的激活、Action 方法的执行以及对请求的响应。毫不夸张地说，整个 MVC 框架是实现在 MvcHandler 之中的。

```
public class MvcRouteHandler : IRouteHandler
{
    //其他成员
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        return new MvcHandler(requestContext)
    }
}
```

2.3.3 ASP.NET 路由系统扩展

到此为止，我们已经对 ASP.NET 的路由系统的实现进行了详细地介绍，总的来说，整个路由系统是通过 IHttpHandler 的动态注册的方式来实现的。具体来说，UrlRoutingModule 通过对代表 Web 应用的 HttpApplication 的 PostResolveRequestCache 事件的注册实现了对请求的拦截。对于被拦截的请求，UrlRoutingModule 利用注册的路由表对其进行匹配和解析，进而得到一个包含所有路由信息的 RouteData 对象，最终借助该对象的 RouteHandler 得到相应的 IHttpHandler，并映射到当前请求。从可扩展性的角度来讲，可以通过如下三种方式来实现我们需要的路由方式。

- 通过集成抽象类 `RouteBase` 创建自定义 `Route` 定制路由逻辑。
- 通过实现接口 `IRouteHandler` 创建自定义 `RouteHandler` 定制 `HttpHandler` 提供机制。
- 通过实现 `IHttpHandler` 创建自定义 `HttpHandler` 来对请求处理作最终的处理。

实例演示：通过自定义 `Route` 对 ASP.NET 路由系统进行扩展（S212）

定义在 ASP.NET 路由系统中默认的路由类型 `Route` 实现了 URL 模式与物理文件之间的映射，如果我们对 WCF REST 有一定的了解，应该知道其中也有类似的实现。具体来说，WCF REST 借助于 `System.UriTemplate` 这个对象实现了同样定义成某个文本模板的 URI 模式与目标操作之间的映射。篇幅所限，我们不能对 WCF REST 的 `UriTemplate` 作详细的介绍，如果读者朋友对此有兴趣可以参阅笔者在 2012 年 4 月出版的《WCF 全面解析》（上、下册）。

我们创建一个新的 ASP.NET Web 应用，并且添加针对程序集 `System.ServiceModel.dll` 的引用（`UriTemplate` 定义在该程序集中），然后创建如下一个针对 `UriTemplate` 的路由类型 `UriTemplateRoute`。

```
public class UriTemplateRoute:RouteBase
{
    public UriTemplate                UriTemplate { get; private set; }
    public IRouteHandler              RouteHandler { get; private set; }
    public RouteValueDictionary       DataTokens { get; private set; }

    public UriTemplateRoute(string template, string physicalPath,
        object dataTokens = null)
    {
        this.UriTemplate = new UriTemplate(template);
        this.RouteHandler = new PageRouteHandler(physicalPath);
        if (null != dataTokens)
        {
            this.DataTokens = new RouteValueDictionary(dataTokens);
        }
        else
        {
            this.DataTokens = new RouteValueDictionary();
        }
    }

    public override RouteData GetRouteData(HttpContextBase httpContext)
    {
        Uri uri = httpContext.Request.Url;
        Uri baseAddress = new Uri(string.Format("{0}://{1}",
            uri.Scheme, uri.Authority));
        UriTemplateMatch match = this.UriTemplate.Match(baseAddress, uri);
        if (null == match)
        {
            return null;
        }
        RouteData routeData = new RouteData();
        routeData.RouteHandler = this.RouteHandler;
        routeData.Route = this;
        foreach (string name in match.BoundVariables.Keys)
```

```

        {
            routeData.Values.Add(name, match.BoundVariables[name]);
        }
        foreach (var token in this.DataTokens)
        {
            routeData.DataTokens.Add(token.Key, token.Value);
        }
        return routeData;
    }

    public override VirtualPathData GetVirtualPath(RequestContext requestContext,
        RouteValueDictionary values)
    {
        Uri uri = requestContext.HttpContext.Request.Url;
        Uri baseAddress = new Uri(string.Format("{0}://{1}",
            uri.Scheme, uri.Authority));
        Dictionary<string, string> variables = new Dictionary<string, string>();
        foreach (var item in values)
        {
            variables.Add(item.Key, item.Value.ToString());
        }

        //确定段变量是否被提供
        foreach (var name in this.UriTemplate.PathSegmentVariableNames)
        {
            if (!this.UriTemplate.Defaults.Keys.Any(
                key => string.Compare(name, key, true) == 0) &&
                !values.Keys.Any(key => string.Compare(name, key, true) == 0))
            {
                return null;
            }
        }

        //确定查询变量是否被提供
        foreach (var name in this.UriTemplate.QueryValueVariableNames)
        {
            if (!this.UriTemplate.Defaults.Keys.Any(
                key => string.Compare(name, key, true) == 0) &&
                !values.Keys.Any(key => string.Compare(name, key, true) == 0))
            {
                return null;
            }
        }

        Uri virtualPath = this.UriTemplate.BindByName(baseAddress, variables);
        string strVirtualPath = virtualPath.ToString().ToLower()
            .Replace(baseAddress.ToString().ToLower(), "");
        VirtualPathData virtualPathData = new VirtualPathData(this,
            strVirtualPath);
        foreach (var token in this.DataTokens)
        {
            virtualPathData.DataTokens.Add(token.Key, token.Value);
        }
        return virtualPathData;
    }
}

```

如上面的代码片段所示, UriTemplateRoute 具有 UriTemplate、DataTokens 和 RouteHandler 三个只读属性, 前两个通过构造函数的参数进行初始化, 后者则是在构造函数中创建的 PageRouteHandler 对象。

在用于对入栈请求进行匹配并获取路由数据的 GetRouteData 方法中, 我们解析出基于应用的基地址并连同请求地址作为参数调用 UriTemplate 的 Match 方法, 如果返回的 UriTemplateMatch 对象不为 Null, 则意味着 URL 模板的模式与请求地址匹配。在匹配的情况下我们创建并返回相应的 RouteData 对象, 否则直接返回 Null。

在用于生成出栈 URL 的 GetVirtualPath 方法中, 通过判断定义在 URL 模板中的变量(包括变量名包含在属性 PathSegmentVariableNames 的路径段变量和包含在 QueryValueVariableNames 属性的查询变量)是否在提供的 RouteValueDictionary 字段或者默认变量列表(通过属性 Defaults 表示)中来确定 URL 模板是否与提供的变量列表匹配。在匹配的情况下通过调用 UriTemplate 的 BindByName 方法得到一个完整的 Uri。由于该方法返回的是相对路径, 所以我们需要将应用基地址剔除并最终创建并返回一个 VirtualPathData 对象。如果不匹配, 则直接返回 Null。

在创建的 Global.asax 文件中采用如下的代码对我们自定义的 UriTemplateRoute 进行注册, 选用的场景还是之前采用的天气预报的例子。我个人觉得基于 UriTemplate 的 URI 模板比针对 Route 的 URL 模板更好用, 其中一点就是它定义默认值的方式更为直接。如下面的代码片段所示, 可以直接将默认值定义在模板中 (“{areacode=010}/{days=2}”)。

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        UriTemplateRoute route = new UriTemplateRoute("{areacode=010}/{days=2}",
            "~/Weather.aspx", new { defaultCity = "BeiJing", defaultDays = 2 });
        RouteTable.Routes.Add("default", route);
    }
}
```

在注册的路由对应的目标页面 Weather.aspx 的后台代码中, 我们定义了如下一个 GenerateUrl 根据指定的区号 (areacode) 和预报天数 (days) 创建一个 Url, 而 Url 的生成直接通过调用 RouteTable 的 Routes 属性的 GetVirtualPathData 方法完成。

```
public partial class Weather : System.Web.UI.Page
{
    public string GenerateUrl(string areacode, int days)
    {
        var values = new { areacode = areacode, days = days };
        RequestContext requestContext = new RequestContext();
        requestContext.HttpContext = new HttpContextWrapper(HttpContext.Current);
        requestContext.RouteData = RouteData;
        return RouteTable.Routes.GetVirtualPath(requestContext,
            new RouteValueDictionary(values)).VirtualPath;
    }
}
```

通过调用 `GenerateUrl` 方法生成的 URL (`areaCode=0512; days=3`) 连同当前页面的 `RouteData` 的属性通过如下所示的 HTML 代码输出出来。

```
<form id="form1" runat="server">
  <div>
    <table>
      <tr>
        <td>Router:</td>
        <td><%=RouteData.Route != null?
          RouteData.Route.GetType().FullName:"" %></td>
      </tr>
      <tr>
        <td>RouteHandler:</td>
        <td><%=RouteData.RouteHandler != null?
          RouteData.RouteHandler.GetType().FullName:"" %></td>
      </tr>
      <tr>
        <td>Values:</td>
        <td>
          <ul>
            <%foreach (var variable in RouteData.Values)
            {%>
              <li>
                <%=variable.Key%>=<%=variable.Value%></li>
            <% }%>
          </ul>
        </td>
      </tr>
      <tr>
        <td>DataTokens:</td>
        <td>
          <ul>
            <%foreach (var variable in RouteData.DataTokens)
            {%>
              <li>
                <%=variable.Key%>=<%=variable.Value%></li>
            <% }%>
          </ul>
        </td>
      </tr>
      <tr>
        <td>Generated Url:</td>
        <td>
          <%=GenerateUrl("0512",3)%>
        </td>
      </tr>
    </table>
  </div>
</form>
```

由于注册的 URL 模板所包含的段均由具有默认值的变量构成, 所以当我们请求根地址时, 会自动路由到 `Weather.aspx`。图 2-14 是我们在浏览器访问应用根目录的截图, 上面显示了我们注册的 `UriTemplateRoute` 生成的 `RouteData` 的信息和生成的 URL (`/0512/3`)。

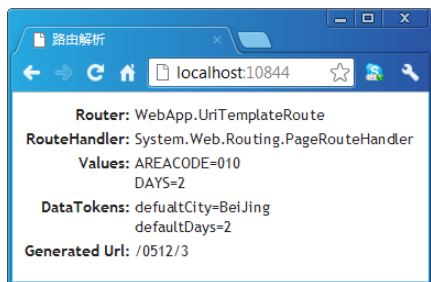


图 2-14 通过自定义 UriTemplateRoute 得到的 RouteData 和生成的 URL

本章小结

ASP.NET MVC 应用下 HTTP 请求的访问目标是定义在某个 Controller 类型中的某个 Action 方法，URL 路由系统通过对请求地址进行解析从而得到以目标 Controller/Action 名称为核心的路由数据。URL 路由系统是建立在 ASP.NET 上，而非专属于 ASP.NET MVC，它最初是为了实现请求 URL 与物理文件路径的分离而建立的。ASP.NET MVC 通过自定义的路由类型实现对 ASP.NET 路由系统的扩展，将 URL 与物理文件路径的映射转移到与目标 Controller/Action 的映射。

ASP.NET 路由系统具有一个针对整个 Web 应用的全局路由表，路由表中的每个路由对象具有一个可以包含变量的 URL 模板。路由对象一方面利用 URL 模板与入栈请求的 URL 进行模式匹配并得到相应的路由数据，另一方面还可以根据指定的路由变量参数列表生成相应的 URL。

ASP.NET 路由系统是通过 HttpHandler 的动态映射来实现的，作为自定义 HttpModule 的 UrlRoutingModule 通过注册 HttpApplication 的 PostResolveRequestCache 事件对请求进行拦截，并利用路由表与请求 URL 进行模式匹配得到相应的路由数据。与请求 URL 相匹配路由对象关联的 HttpHandler 被提取出来用于最终处理当前请求。

第3章 Controller 的激活

ASP.NET MVC 应用中请求的目标不再是具体某个物理文件,而是定义在某个 Controller 中的 Action 方法。每个请求经过 ASP.NET URL 路由系统的拦截后,会生成以 Controller/Action 名称为核心的路由数据。ASP.NET MVC 据此解析出目标 Controller 的类型,并最终激活具体的 Controller 实例来处理当前的请求。

3.1 总体设计

我们将整个 ASP.NET MVC 框架人为地划分为若干个子系统，那么针对请求上下文激活目标 Controller 对象的子系统可以称为 Controller 激活系统。在正式讨论 Controller 对象具体是如何被创建之前，我们先来了解 Controller 激活系统在 ASP.NET MVC 中的总体设计，看看它大体上由哪些组件构成。

3.1.1 Controller

我们知道作为 Controller 的类型直接或者间接实现了 `System.Web.Mvc.IController` 接口。如下面的代码片段所示，`IController` 接口仅仅包含一个参数类型为 `RequestContext` 的 `Execute` 方法，当一个 Controller 对象被激活之后，其核心的操作就是：从包含在当前请求上下文的路由数据中获取 Action 名称并据此解析出对应的方法，将通过 Model 绑定机制从当前请求上下文中提取相应的数据并调用 Action 方法生成对应的参数列表。所有这些后续操作都是间接地通过调用 Controller 的 `Execute` 方法来完成的。

```
public interface IController
{
    void Execute(RequestContext requestContext);
}
```

定义在 `IController` 接口中的 `Execute` 是以同步的方式执行的。为了支持以异步方式对请求的处理，`IController` 接口的异步版本 `System.Web.Mvc.IAsyncController` 被定义出来。如下面的代码片段所示，实现了 `IAsyncController` 接口 Controller 的执行通过 `BeginExecute/EndExecute` 方法以异步的形式完成。

```
public interface IAsyncController : IController
{
    IAsyncResult BeginExecute(RequestContext requestContext,
        AsyncCallback callback, object state);
    void EndExecute(IAsyncResult asyncResult);
}
```

抽象类 `System.Web.Mvc.ControllerBase` 实现了 `IController` 接口。如下面的代码片段所示，`ControllerBase` 以“显式接口实现”的方式定义了 `Execute` 方法，该方法在内部直接调用受保护的 `Execute` 虚方法，而后者最终会调用抽象方法 `ExecuteCore` 方法。

```
public abstract class ControllerBase : IController
{
    //其他成员
    protected virtual void Execute(RequestContext requestContext);
    protected abstract void ExecuteCore();
    void IController.Execute(RequestContext requestContext);

    public ControllerContext ControllerContext { get; set; }
```

```

public TempDataDictionary    TempData { get; set; }
public object                ViewBag { [return: Dynamic] get; }
public ViewDataDictionary    ViewData { get; set; }
}

```

ControllerBase 具有如下几个重要的属性：TempData、ViewBag 和 ViewData，它们用于存储从 Controller 向 View 传递的数据或者变量。其中 TempData 和 ViewData 具有基于字典的数据结构，Key 和 Value 分别表示变量的名称和值，两者的不同之处在于前者仅仅用于存储临时数据，并且设置的变量被第一次读取之后会被移除，换句话说通过 TempData 设置的变量只能被读取一次。ViewBag 和 ViewData 共享着相同的数据，它们之间的不同之处在于前者是一个动态对象，我们可以为其指定任意属性（动态属性名将作为数据字典的 Key）。

在 ASP.NET MVC 中我们会陆续遇到一系列的上下文（Context）对象，之前已经对表示请求上下文的 RequestContext（HttpContext + RouteData）进行了详细的介绍，现在来介绍另一个具有如下定义的上下文类型 System.Web.Mvc.ControllerContext。

```

public class ControllerContext
{
    //其他成员
    public ControllerContext();
    public ControllerContext(RequestContext requestContext,
        ControllerBase controller);
    public ControllerContext(HttpContextBase httpContext,
        RouteData routeData, ControllerBase controller);

    public virtual ControllerBase    ControllerBase { get; set; }
    public virtual RequestContext     RequestContext { get; set; }
    public virtual HttpContextBase    HttpContextBase { get; set; }
    public virtual RouteData          RouteData { get; set; }
}

```

顾名思义，ControllerContext 就是基于某个 Controller 对象的上下文。从如上的代码可以看出一个 ControllerContext 对象实际上是对一个 Controller 对象和 RequestContext 的封装。这两个对象分别对应着 ControllerContext 中的同名属性，可以在构建 ControllerContext 的时候为调用的构造函数指定相应的参数来初始化它们。

通过 HttpContext 和 RouteData 属性返回的 HttpContextBase 和 RouteData 对象在默认情况下实际上就是 RequestContext 的核心组成部分。当 ControllerBase 的 Execute 方法被执行的时候，它会根据传入的 RequestContext 创建 ControllerContext 对象，后续的操作可以看成是在该上下文中进行。

通过 Visual Studio 的 Controller 创建向导创建的 Controller 类型实际上继承自抽象类 System.Web.Mvc.Controller，它是 ControllerBase 的子类。如下面的代码片段所示，除了直接继承 ControllerBase 之外，Controller 类型还显式地实现了 IController 和 IAsyncController 接口，以及代表 ASP.NET MVC 四大筛选器（AuthorizationFilter、ActionFilter、ResultFilter 和 ExceptionFilter）的 4 个接口（我们会在第 7 章“Action 的执行”中对筛选器进行详细介绍）。

```
public abstract class Controller :
    ControllerBase,
    IController,
    IAsyncController,
    IActionFilter,
    IAuthorizationFilter,
    IExceptionFilter,
    IResultFilter,
    IDisposable,
    ...
{
    //省略成员
}
```

同步还是异步

从抽象类 `Controller` 的定义可以看出它同时实现了 `IController` 和 `IAsyncController` 这两个接口，意味着它既可以采用同步的方式（调用 `Execute` 方法）执行，也可以采用异步的方式（调用 `BeginExecute/EndExecute` 方法）执行。但是即使执行 `BeginExecute/EndExecute` 方法，`Controller` 也不一定是以异步方式执行的。

如下面的代码片段所示，`Controller` 具有一个布尔类型的属性 `DisableAsyncSupport`，表示是否关闭对异步执行的支持。在默认的情况下该属性总是返回 `False`，即支持以异步方式执行 `Controller`。`BeginExecute` 方法会根据 `DisableAsyncSupport` 属性决定究竟是调用 `Execute` 方法以同步的方式执行，还是调用 `BeginExecuteCore/EndExecuteCore` 方法以异步的方式执行。换句话说，如果我们希望 `Controller` 总是以同步的方式来执行，可以将 `DisableAsyncSupport` 属性设置为 `True`。

```
public abstract class Controller: ...
{
    //其他成员
    protected virtual bool DisableAsyncSupport
    {
        get{return false;}
    }

    protected virtual IAsyncResult BeginExecute(RequestContext requestContext,
        AsyncCallback callback, object state)
    {
        if (this.DisableAsyncSupport)
        {
            //通过调用 Execute 方法同步执行 Controller
        }
        else
        {
            //通过调用 BeginExecuteCore/EndExecuteCore 方法异步执行 Controller
        }
    }
    protected virtual IAsyncResult BeginExecuteCore(AsyncCallback callback,
        object state);
    protected virtual void EndExecuteCore(IAsyncResult asyncResult);
}
```

现在我们通过一个简单的实例来演示属性 `DisableAsyncSupport` 对默认创建的 `Controller` 执行的影响。我们在一个 `ASP.NET MVC` 应用中定义了一个具有如下定义的默认 `HomeController`，它重写了 `Execute`、`ExecuteCore`、`BeginExecute/EndExecute` 和 `BeginExecuteCore/EndExecuteCore` 六个方法，同时将相应的方法名写入响应并最终呈现在浏览器上。

```
public class HomeController : Controller
{
    public new HttpResponse Response
    {
        get { return System.Web.HttpContext.Current.Response; }
    }

    protected override void Execute(RequestContext requestContext)
    {
        Response.Write("Execute(); <br/>");
        base.Execute(requestContext);
    }

    protected override void ExecuteCore()
    {
        Response.Write("ExecuteCore(); <br/>");
        base.ExecuteCore();
    }

    protected override IAsyncResult BeginExecute(RequestContext requestContext,
        AsyncCallback callback, object state)
    {
        Response.Write("BeginExecute(); <br/>");
        return base.BeginExecute(requestContext, callback, state);
    }

    protected override void EndExecute(IAsyncResult asyncResult)
    {
        Response.Write("EndExecute(); <br/>");
        base.EndExecute(asyncResult);
    }

    protected override IAsyncResult BeginExecuteCore(AsyncCallback callback,
        object state)
    {
        Response.Write("BeginExecuteCore(); <br/>");
        return base.BeginExecuteCore(callback, state);
    }

    protected override void EndExecuteCore(IAsyncResult asyncResult)
    {
        Response.Write("EndExecuteCore(); <br/>");
        base.EndExecuteCore(asyncResult);
    }

    public ActionResult Index()
    {
        return Content("Index();<br/>");
    }
}
```

虽然抽象类中定义了一个表示当前 `HttpResponse` 的属性 `Response`，但是当 `BeginExecute`

方法执行的时候该属性尚未初始化，所以上面代码中使用的 `Response` 属性是我们自行定义的。运行该程序后会在浏览器中呈现出如图 3-1 所示的输出结果。从输出方法的调用顺序中不难看出在默认的情况下 `Controller` 是以异步的方式执行的。（S301）

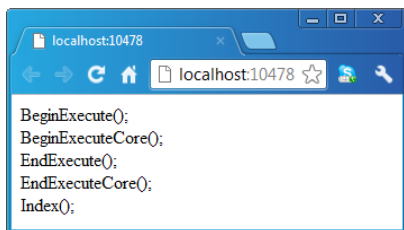


图 3-1 `Controller` 在默认情况下的异步执行方式

现在按照如下的方式重写虚属性 `DisableAsyncSupport`，使它直接返回 `True` 以关闭对 `Controller` 异步执行的支持。

```
public class HomeController : Controller
{
    //其他成员
    protected override bool DisableAsyncSupport
    {
        get{return true;}
    }
}
```

再次执行我们的程序将会得到如图 3-2 所示的输出结果，可以看出由于 `HomeController` 间接地实现了 `IAsyncController` 接口，`Controller` 的执行总是以调用 `BeginExecute/EndExecute` 方法的方式来执行，但是由于 `DisableAsyncSupport` 属性被设置为 `True`，`BeginExecute` 方法内部会以同步的方式调用 `Execute/ExecuteCore` 方法。（S302）

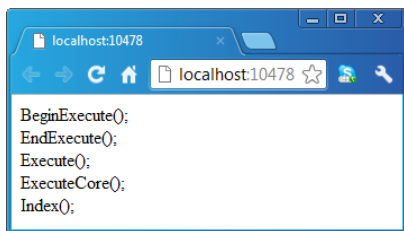


图 3-2 `Controller` 在 `DisableAsyncSupport` 属性为 `True` 的情况下的同步执行方式

ASP.NET MVC 应用编程接口中还定义了一个 `System.Web.Mvc.AsyncController` 类型，从名称上看，`AsyncController` 是一个基于异步的 `Controller`，但是这里的异步并不是指 `Controller` 的异步执行，而是 `Action` 方法的异步执行。从如下的代码片段中可以看出，这个直接继承自抽象类 `Controller` 的 `AsyncController` 是一个“空”类型（没有额外定义和重写基类的类型成员）。在上一个版本中，以 `XxxAsync/XxxCompleted` 形式定义的异步 `Action` 方法均定义在

继承自 `AsyncController` 的 `Controller` 类型中，考虑到向后兼容性，`AsyncController` 在新的版本中保留下来。

```
public abstract class AsyncController : Controller
{ }
```

只有以传统方式（`XxxAsync/XxxCompleted`）定义的异步 `Action` 方法才需要定义在 `AsyncController` 中。ASP.NET MVC 4.0 提供了新的异步 `Action` 方法定义方式，使我们可以通过一个返回类型为 `Task` 的方法来定义以异步方式执行的 `Action`，这样的 `Action` 方法不需要定义在 `AsyncController` 中。

3.1.2 ControllerFactory

ASP.NET MVC 为 `Controller` 的激活定义相应的工厂，我们将其统称为 `ControllerFactory`，所有的 `ControllerFactory` 实现了接口 `System.Web.Mvc.IControllerFactory` 接口。如下面的代码片段所示，`Controller` 对象的激活最终通过 `IControllerFactory` 的 `CreateController` 方法来完成，该方法两个参数分别表示当前请求上下文和从路由信息中获取的 `Controller` 的名称。

```
public interface IControllerFactory
{
    IController CreateController(RequestContext requestContext,
        string controllerName);
    SessionStateBehavior GetControllerSessionBehavior(
        RequestContext requestContext, string controllerName);
    void ReleaseController(IController controller);
}

public enum SessionStateBehavior
{
    Default,
    Required,
    ReadOnly,
    Disabled
}
```

除了负责创建 `Controller` 处理请求之外，`ControllerFactory` 还需要在完成请求处理之后释放 `Controller`，对激活 `Controller` 对象的释放定义在 `ReleaseController` 方法中。`IControllerFactory` 的另一个方法 `GetControllerSessionBehavior` 返回一个 `System.Web.SessionState.SessionStateBehavior` 枚举。熟悉 ASP.NET 的读者对 `SessionStateBehavior` 应该不会感到陌生，它用于表示请求处理过程中会话状态支持的模式，它的四个枚举值分别具有如下的含义。

- **Default**: 使用默认 ASP.NET 逻辑来确定请求的会话状态行为。
- **Required**: 为请求启用完全的读写会话状态行为。
- **ReadOnly**: 为请求启用只读会话状态。
- **Disabled**: 禁用会话状态。

对于 Default 选项来说, ASP.NET 通过映射的 `HttpHandler` 类型是否实现了相关接口来决定具体的会话状态控制行为。在 `System.Web.SessionState` 命名空间下定义了 `IRequiresSessionState` 和 `IReadOnlySessionState` 接口, 如下面的代码片段所示, 这两个都是不具有任何成员的空接口 (我们一般称之为标记接口), 而 `IReadOnlySessionState` 继承自 `IRequiresSessionState`。如果 `HttpHandler` 实现了接口 `IReadOnlySessionState`, 则意味着采用 `ReadOnly` 模式, 如果只实现了 `IRequiresSessionState` 则采用 `Required` 模式。

```
public interface IRequiresSessionState
{
}

public interface IReadOnlySessionState : IRequiresSessionState
{
}
```

具体采用何种会话状态行为取决于当前 HTTP 上下文 (通过 `HttpContext` 的静态属性 `Current` 表示)。对于之前的版本, 我们不能对当前 HTTP 上下文的会话状态行为模式进行动态的修改, ASP.NET 4.0 为 `HttpContext` 定义了如下一个 `SetSessionStateBehavior` 方法使我们可以自由地选择会话状态行为模式。相同的方法同样定义在 `HttpContextBase` 中, 它的子类 `HttpContextWrapper` 重写了这个方法并在内部会调用封装的 `HttpContext` 的同名方法。

```
public sealed class HttpContext : IServiceProvider, IPrincipalContainer
{
    //其他成员
    public void SetSessionStateBehavior(
        SessionStateBehavior sessionStateBehavior);
}

public class HttpContextBase: IServiceProvider
{
    //其他成员
    public void SetSessionStateBehavior(
        SessionStateBehavior sessionStateBehavior);
}
```

3.1.3 ControllerBuilder

用于激活 `Controller` 对象的 `ControllerFactory` 最终通过 `System.Web.Mvc.ControllerBuilder` 注册到 ASP.NET MVC 应用中。如下面的代码所示, `ControllerBuilder` 定义了一个静态只读属性 `Current` 返回当前 `ControllerBuilder` 对象, 这是针对整个 Web 应用的全局对象。两个 `SetControllerFactory` 方法重载用于注册 `ControllerFactory` 的类型或者实例, 而 `GetControllerFactory` 方法返回一个具体的 `ControllerFactory` 对象。

```
public class ControllerBuilder
{
    public IControllerFactory GetControllerFactory();

    public void SetControllerFactory(Type controllerFactoryType);
}
```



```

public void SetControllerFactory(IControllerFactory controllerFactory);

public HashSet<string> DefaultNamespaces { get; }
public static ControllerBuilder Current { get; }
}

```

具体来说，如果我们注册的是 `ControllerFactory` 的类型，那么 `GetControllerFactory` 在运行的时候会通过对注册类型的反射（调用 `Activator` 的静态方法 `CreateInstance`）来创建具体的 `ControllerFactory`（系统不会对创建的 `Controller` 进行缓存）。如果注册的是一个具体的 `ControllerFactory` 对象，该对象直接从 `GetControllerFactory` 返回。

通过第2章“URL 路由”的介绍我们知道，被 ASP.NET 路由系统进行拦截处理后会生成一个用于封装路由信息的 `RouteData` 对象，而目标 `Controller` 的名称就包含在通过该 `RouteData` 的 `Values` 属性表示的 `RouteValueDictionary` 对象中，对应的 `Key` 为“controller”。而在默认的情况下，这个作为路由数据的名称只能帮助我们解析出 `Controller` 的类型名称，如果在不同的命名空间下定义了多个同名的 `Controller` 类，会导致激活系统无法确定具体的 `Controller` 的类型从而抛出异常。

为了解决这个问题，我们必须为定义了同名 `Controller` 类型的命名空间设置不同的优先级。具体来说有两种提升命名空间优先级的方式。第一种方式就是在调用 `RouteCollection` 如下所示的扩展方法 `MapRoute` 时指定一个命名空间的列表。通过第2章“URL 路由”的介绍我们知道，通过这种方式指定的命名空间列表会保存在 `Route` 对象的 `DataTokens` 属性表示的 `RouteValueDictionary` 字典中，对应的 `Key` 为“Namespaces”。

```

public static class RouteCollectionExtensions
{
    //其他成员
    public static Route MapRoute(this RouteCollection routes, string name,
        string url, string[] namespaces);
    public static Route MapRoute(this RouteCollection routes, string name,
        string url, object defaults, string[] namespaces);
    public static Route MapRoute(this RouteCollection routes, string name,
        string url, object defaults, object constraints, string[] namespaces);
}

```

另一种提升命名空间优先级的方式就是将其添加到当前的 `ControllerBuilder` 中的默认命名空间列表中。从上面的给出的 `ControllerBuilder` 的定义可以看出，它具有一个 `HashSet<string>` 类型的只读属性 `DefaultNamespaces` 代表了这么一个默认命名空间列表。对于这两种不同的命名空间优先级提升方式，前者（通过路由注册）指定命名空间具有更高的优先级。

实例演示：如何提升命名空间的优先级（S303，S304，S305）

为了让读者对如何提升命名空间优先级有一个深刻的印象，我们来进行一个简单的实例演示。在一个 ASP.NET MVC 应用创建两个同名的 `HomeController` 类，如下面的代码片段所示，这两个 `HomeController` 类分别定义在命名空间 `Artech.MvcApp` 和 `Artech.MvcApp`。

Controllers 之中，而 Index 操作返回的是一个将 Controller 类型全名作为内容的 System.Web.Mvc.ContentResult 对象。

```
namespace Artech.MvcApp.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return this.Content(this.GetType().FullName);
        }
    }
}

namespace Artech.MvcApp
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return this.Content(this.GetType().FullName);
        }
    }
}
```

现在我们直接运行该 Web 应用。由于具有多个 Controller 与注册的路由规则相匹配，这会导致 Controller 激活系统无法确定哪个类型的 Controller 应该被选用，所以会出现如图 3-3 所示的错误。（S303）

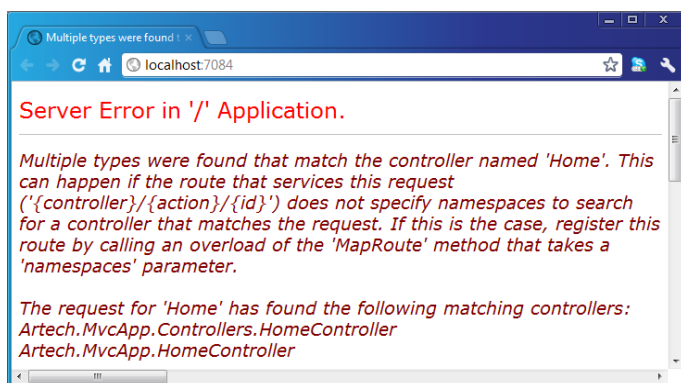


图 3-3 具有多个匹配 Controller 导致的异常

目前定义了 HomeController 的两个命名空间具有相同的优先级，现在将其中一个定义在当前 ControllerBuilder 的默认命名空间列表中以提升匹配优先级。如下面的代码片段所示，在 Global.asax 的 Application_Start 方法中，将命名空间“Artech.MvcApp.Controllers”添加到当前 ControllerBuilder 的 DefaultNamespaces 属性所示的命名空间列表中。

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        ControllerBuilder.Current.DefaultNamespaces
            .Add("Artech.MvcApp.Controllers");
    }
}
```

对于同时匹配注册的路由规则的两个 HomeController 来说, 由于 “Artech.MvcApp.Controllers” 命名空间具有更高的匹配优先级, 所有定义其中的 HomeController 会被选用, 这可以通过如图 3-4 所示的运行结果看出来。(S304)

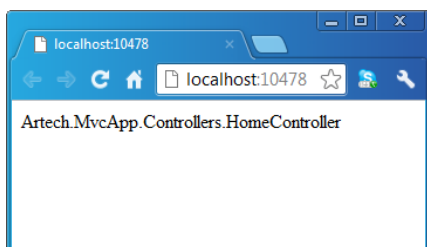


图 3-4 通过 ControllerBuilder 提升命名空间匹配优先级

为了检验在路由注册时指定的命名空间和作为当前 ControllerBuilder 的命名空间哪个具有更高匹配优先级, 修改定义在 “App_Start/RouteConfig.cs” 中的路由注册代码, 如下面的代码片段所示, 在调用 RouteTable 的静态属性 Routes 的 MapRoute 方法进行路由注册的时候指定了命名空间 (“Artech.MvcApp”)。

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        //其他操作
        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index",
                           id = UrlParameter.Optional },
            namespaces: new string[] { "Artech.MvcApp" }
        );
    }
}
```

再次运行我们的程序会在浏览器中得到如图 3-5 所示的结果, 从中可以看出定义在命名空间 “Artech.MvcApp” 中的 HomeController 被最终选用, 可见较之作为当前 ControllerBuilder 的默认命名空间, 在路由注册过程中执行的命名空间具有更高的匹配优先级, 前者可以视为后者的一种后备。(S305)

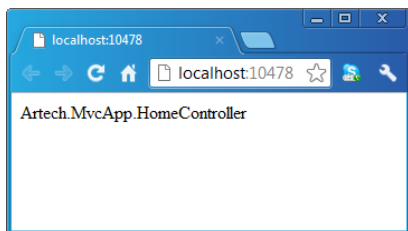


图 3-5 在路由注册时指定的命名空间具有更高的匹配优先级

在路由注册时指定的命名空间比当前 `ControllerBuilder` 的默认命名空间具有更高的匹配优先级，但是对于这两个集合中的所有命名空间却具有相同的匹配优先级。换句话说，用于辅助解析 `Controller` 类型的命名空间分为三个梯队，分别简称为路由命名空间、`ControllerBuilder` 命名空间和 `Controller` 类型命名空间。如果前一个梯队不能正确解析出目标 `Controller` 的类型，则后一个梯队的命名空间将作为后备，反之，如果根据某个梯队的命名空间进行解析得到多个匹配的 `Controller` 类型，会直接抛出异常。

针对 Area 的路由对象的命名空间

针对某个 Area 的路由映射是通过相应的 `AreaRegistration` 进行注册的，具体来说是在 `AreaRegistration` 的 `RegisterArea` 方法中调用 `AreaRegistrationContext` 对象的 `MapRoute` 方法进行注册的。如果在调用 `MapRoute` 方法中指定了表示命名空间的字符串，它将自动作为注册的路由对象的命名空间，否则会将 `AreaRegistration` 的命名空间加上 “.” 后缀得到的字符串作为路由对象的命名空间。

这里所说的“路由对象的命名空间”存在于 `Route` 对象的 `DataTokens` 属性表示的 `RouteValueDictionary` 对象中，对应的 Key 为 “Namespaces”，Value 就是一个包含字符串数组的命名空间列表。通过第 2 章“URL 路由”的介绍，`Route` 对象的 `DataTokens` 属性包含的变量会转移到由它生成的 `RouteData` 的同名属性中。

除此之外，在调用 `AreaRegistrationContext` 的 `MapRoute` 方法时还会在注册 `Route` 对象的 `DataTokens` 属性中添加一个 Key 为 “UseNamespaceFallback” 的条目，它表示是否采用后备命名空间对 `Controller` 类型进行解析。如果注册的路由对象具有命名空间（调用 `MapRoute` 方法时指定了命名空间或者对应的 `AreaRegistration` 类型定义在某个命名空间下），该条目的值为 `False`，否则为 `True`。该条目同样反映在通过该 `Route` 对象生成的 `RouteData` 对象的 `DataTokens` 属性中。

在解析 `Controller` 真实类型的过程中，会先使用 `RouteData` 包含的命名空间。如果解析失败，则通过由 `RouteData` 的 `DataTokens` 属性得到的这个名为 “UseNamespaceFallback” 的变量值来判断是否使用“后备”命名空间进行解析。具体来说，如果该值为 `True` 或者不存在，则先通过当前 `ControllerBuilder` 的命名空间解析，如果失败则忽略命名空间直接采用类型名称进行匹配，否则会因找不到匹配的 `Controller` 而直接抛出异常。

我们通过具体的例子来说明这个问题。在一个 ASP.NET MVC 应用中通过 Area 添加向导创建一个名称为 Admin 的 Area，此时 IDE 会默认为我们添加了如下一个 `AdminAreaRegistration` 类型。

```

NamespaceMvcApp.Areas.Admin
{
    public class AdminAreaRegistration : AreaRegistration
    {
        public override string AreaName
        {
            get{return "Admin";}
        }
        public override void RegisterArea(AreaRegistrationContext context)
        {
            context.MapRoute(
                "Admin_default",
                "Admin/{controller}/{action}/{id}",
                new { action = "Index", id = UrlParameter.Optional }
            );
        }
    }
}

```

`AdminAreaRegistration` 类型定义在命名空间 `MvcApp.Areas.Admin` 中。现在我们在该 Area 中添加如下一个 `HomeController`，在默认的动作方法 `Index` 中，我们从当前 `RouteData` 的 `DataTokens` 中提取这个名为“`UseNamespaceFallback`”的变量值，并将它和解析出来的 Controller 类型名称写入当前 `HttpResponse` 而最终呈现在客户端浏览器中。在默认情况下，添加的 `HomeController` 类型被定义在 `MvcApp.Areas.Admin.Controllers` 命名空间下，现在我们刻意将命名空间改为 `MvcApp.Areas.Controllers`。

```

namespaceMvcApp.Areas.Controllers
{
    public class HomeController : Controller
    {
        public void Index()
        {
            Response.Write(string.Format("UseNamespaceFallback: {0}<br/>",
                RouteData.DataTokens["UseNamespaceFallback"]));
            Response.Write(string.Format("Controller Type: {0}<br/>",
                this.GetType().FullName));
        }
    }
}

```

现在我们在浏览器中通过匹配的 URL (`/Admin/Home/Index`) 来访问 Area 为 Admin 的 `HomeController` 的 `Index` 操作，会得到如图 3-6 所示的 HTTP 状态为“404, Not Found”的错误。这就是因为在对 Controller 类型进行解析的时候是严格按照对应的 `AreaRegistration` 所在的命名空间来进行的，很显然在这个范围内是不可能找得到对应的 Controller 类型的。
(S306)

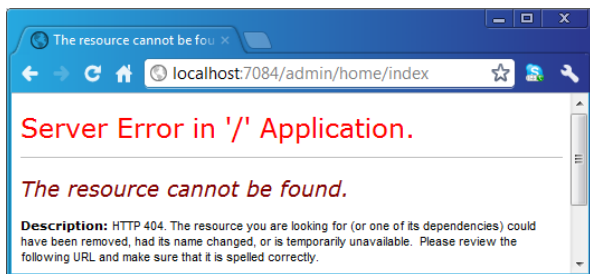


图 3-6 Controller 和 AreaRegistration 命名空间不匹配导致的 404 错误

但是如果我们去掉 `AdminAreaRegistration` 的命名空间，那么将会导致路由变量 `UseNamespaceFallback` 的值变为 `True`，这会促使 Controller 激活系统选择“后备”的命名空间。由于整个 Web 应用中仅仅定义了唯一匹配的 `MvcApp.Areas.Controllers.HomeController`，很显然这个 Controller 会被激活，如图 3-7 所示的程序运行结果也说明了这一点。（S307）

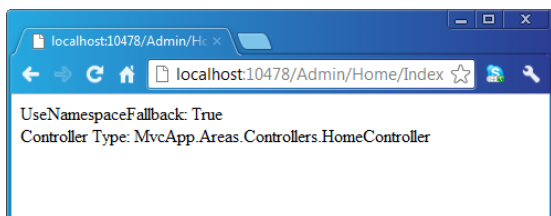


图 3-7 去掉 `AdminAreaRegistration` 命名空间以采用后备命名空间

3.1.4 Controller 的激活与 URL 路由

ASP.NET 路由系统是 HTTP 请求抵达服务端的第一道屏障，它根据注册的路由规则对拦截的请求进行匹配并解析包含目标 Controller 和 Action 名称的路由信息。而当前 `ControllerBuilder` 具有用于激活 Controller 对象的 `ControllerFactory`，现在看看两者是如何结合起来的。

通过第 2 章“URL 路由”的介绍我们知道，ASP.NET 路由系统的核心是一个叫做 `UrlRoutingModule` 的 `HttpModule`，路由的实现是它通过注册代表 `HttpApplication` 的 `PostResolveRequestCache` 事件对 `HttpHandler` 的动态映射来实现的。具体来说，它通过以 `RouteTable` 的静态属性 `Routes` 代表的全局路由表对请求进行匹配并得到一个 `RouteData` 对象。`RouteData` 具有一个实现了接口 `IRouteHandler` 的属性 `RouteHandler`，通过该属性的 `GetHttpHandler` 方法可以得到最终被映射到当前请求的 `HttpHandler` 对象。

对于 ASP.NET MVC 应用来说，`RouteData` 的 `RouteHandler` 属性类型为 `MvcRouteHandler`，实现在 `MvcRouteHandler` 中的 `HttpHandler` 提供机制基本上（不是完全等同）可以通过如下代码来体现。`MvcRouteHandler` 维护着一个 `ControllerFactory` 对象，该对象可以在构造函数

数中指定，如果没有显示指定则直接通过调用当前 `ControllerBuilder` 的 `GetControllerFactory` 方法获取。

```
public class MvcRouteHandler : IRouteHandler
{
    private IControllerFactory _controllerFactory;
    public MvcRouteHandler(): this(ControllerBuilder.Current
        .GetControllerFactory())
    { }
    public MvcRouteHandler(IControllerFactory controllerFactory)
    {
        _controllerFactory = controllerFactory;
    }
    IHttpHandler IRouteHandler.GetHttpHandler(RequestContext requestContext)
    {
        string controllerName = (string)requestContext.RouteData
            .GetRequiredString("controller");
        SessionStateBehavior sessionStateBehavior = _controllerFactory
            .GetControllerSessionBehavior(requestContext, controllerName);
        requestContext.HttpContext.SetSessionStateBehavior(sessionStateBehavior);

        return new MvcHandler(requestContext);
    }
}
```

在用于提供 `HttpHandler` 的 `GetHttpHandler` 方法中，除了返回一个实现了 `IHttpHandler` 接口的 `MvcHandler` 对象之外，还需要对当前 HTTP 上下文的会话状态行为模式进行设置。具体的实现是：先通过包含在 `RequestContext` 的 `RouteData` 对象得到 `Controller` 的名称，该名称连同 `RequestContext` 对象一起传入 `ControllerFactory` 的 `GetControllerSessionBehavior` 方法得到一个类型为 `SessionStateBehavior` 的枚举。最后通过 `RequestContext` 得到当前 HTTP 上下文（实际上是一个 `HttpContextWrapper` 对象），并调用其 `SetSessionStateBehavior` 方法对会话状态行为进行设置。

通过第2章“URL 路由”的介绍我们知道，`RouteData` 中的 `RouteHandler` 属性最初来源于对应的路由对象，而当我们调用 `RouteCollection` 的扩展方法 `MapRoute` 方法时注册的 `Route` 对象对应的 `RouteHandler` 是一个 `MvcRouteHandler` 对象。由于在创建 `MvcRouteHandler` 对象时并没有显式指定 `ControllerFactory`，所以通过当前 `ControllerBuilder` 的 `GetControllerFactory` 方法得到的 `ControllerFactory` 默认被使用。

通过当前 `ControllerBuilder` 的 `GetControllerFactory` 方法得到的 `ControllerFactory` 仅仅用于获取会话状态行为模式，而 `MvcHandler` 真正将它用于创建 `Controller`。如下的代码片段基本上体现了 `MvcHandler` 的定义，它对请求处理的逻辑定义在 `BeginProcessRequest` 方法中。

```
public class MvcHandler : IHttpAsyncHandler, IHttpHandler, IRequiresSessionState
{
    //其他成员
    public RequestContext RequestContext { get; private set; }

    public bool IsReusable
    {
```

```

        get { return false; }
    }

    public MvcHandler(RequestContext requestContext)
    {
        this.RequestContext = requestContext;
    }

    IAsyncResult BeginProcessRequest(HttpContext context, AsyncCallback cb,
        object extraData)
    {
        IControllerFactory controllerFactory =
            ControllerBuilder.Current.GetControllerFactory();
        string controllerName =
            this.RequestContext.RouteData.GetRequiredString("controller");
        IController controller = controllerFactory
            .CreateController(this.RequestContext, controllerName);
        if (controller is IAsyncController)
        {
            {
                try
                {
                    //调用 BeginExecute/EndExecute 方法以异步的方式执行 Controller
                }
                finally
                {
                    controllerFactory.ReleaseController(controller);
                }
            }
        }
        else
        {
            {
                try
                {
                    //调用 Execute 方法以异步的方式执行 Controller
                }
                finally
                {
                    controllerFactory.ReleaseController(controller);
                }
            }
        }
    }
}

```

由于 `MvcHandler` 同时实现了 `IHttpHandler` 和 `IHttpAsyncHandler` 接口，所以它总是以异步的方式被执行（调用 `BeginProcessRequest/EndProcessRequest` 方法）。`BeginProcessRequest` 方法通过 `RequestContext` 对象得到目标 `Controller` 的名称，然后利用当前 `ControllerBuilder` 创建的 `ControllerFactory` 激活 `Controller` 对象。如果 `Controller` 类型实现了 `IAsyncController` 接口，则以异步的方式执行 `Controller`，否则采用同步执行方式。在被激活 `Controller` 对象被执行之后，`MvcHandler` 会调用 `ControllerFactory` 的 `ReleaseController` 对其进行释放清理工作。

3.2 默认实现

`Controller` 激活系统最终通过注册的 `ControllerFactory` 创建相应的 `Controller` 对象，如果

没有对 `ControllerFactory` 类型或者实例进行注册（通过调用当前 `ControllerBuilder` 的 `SetControllerFactory` 方法），默认使用的 `ControllerFactory` 类型为 `System.Web.Mvc.DefaultControllerFactory`。我们现在就来讨论实现在 `DefaultControllerFactory` 中的默认 Controller 激活机制。

3.2.1 Controller 类型的解析

激活目标 Controller 对象的前提是能够正确解析出 Controller 类型。对于 `DefaultControllerFactory` 来说，用于解析目标 Controller 类型的辅助信息包括：通过与当前请求匹配的路由对象生成的 `RouteData`（其中包含 Controller 的名称和命名空间）和包含在当前 `ControllerBuilder` 中的命名空间。很多读者可能首先想到的是通过 Controller 名称得到对应的类型，并通过命名空间组成 Controller 类型的全名，最后遍历所有程序集并以此名称去加载相应的类型即可。

这貌似是一个不错的解决方案，实际上则完全行不通。不要忘了作为请求地址 URL 一部分的 Controller 名称是不区分大小写的，而类型名称则是大小写敏感的。此外，不论是注册路由时指定的命名空间还是当前 `ControllerBuilder` 的默认命名空间，有可能包含统配符（*）。由于我们不能通过给定的 Controller 名称和命名空间得到 Controller 的真实类型名称，自然就不可能通过名称去解析 Controller 的类型了。

ASP.NET MVC 的 Controller 激活系统则反其道而行之，它先遍历通过 `BuildManager` 的静态方法 `GetReferencedAssemblies` 方法得到所有引用程序集，通过反射的方式得到定义在它们中的所有实现了接口 `IController` 的类型，最后通过 Controller 的名称和命名空间作为匹配条件去选择对应的 Controller 类型。

实例演示：创建一个自定义 ControllerFactory 模拟 Controller 默认激活机制（S308）

为了让读者对默认采用的 Controller 激活机制，尤其是 Controller 类型的解析机制有一个深刻的认识，通过一个自定义的 `ControllerFactory` 来模拟其中的实现。由于采用反射的方式来创建 Controller 对象，所以将该自定义 `ControllerFactory` 起名为 `ReflectedControllerFactory`。

```
public class ReflectedControllerFactory : IControllerFactory
{
    //其他成员
    private static List<Type> controllerTypes;
    static ReflectedControllerFactory()
    {
        controllerTypes = new List<Type>();
        foreach (Assembly assembly in BuildManager.GetReferencedAssemblies())
        {
            controllerTypes.AddRange(assembly.GetTypes().Where(
                type => typeof(IController).IsAssignableFrom(type)));
        }
    }
}
```

```

public IController CreateController(RequestContext requestContext,
    string controllerName)
{
    Type controllerType = this.GetControllerType(requestContext.RouteData,
        controllerName);
    if (null == controllerType)
    {
        return null;
    }
    return (IController)Activator.CreateInstance(controllerType);
}

private static bool IsNamespaceMatch(string requestedNamespace,
    string targetNamespace)
{
    if (!requestedNamespace.EndsWith(".*",
        StringComparison.OrdinalIgnoreCase))
    {
        return string.Equals(requestedNamespace, targetNamespace,
            StringComparison.OrdinalIgnoreCase);
    }
    requestedNamespace = requestedNamespace.Substring(0,
        requestedNamespace.Length - ".*".Length);
    if (!targetNamespace.StartsWith(requestedNamespace,
        StringComparison.OrdinalIgnoreCase))
    {
        return false;
    }
    return ((requestedNamespace.Length == targetNamespace.Length) ||
        (targetNamespace[requestedNamespace.Length] == '.'));
}

private Type GetControllerType(IEnumerable<string> namespaces,
    Type[] controllerTypes)
{
    var types = (from type in controllerTypes
        where namespaces.Any(ns => IsNamespaceMatch(
            ns, type.Namespace))
        select type).ToArray();
    switch (types.Length)
    {
        {
            case 0: return null;
            case 1: return types[0];
            default: throw new InvalidOperationException("具有多个匹配的 Controller
                类型");
        }
    }
}

protected virtual Type GetControllerType(RouteData routeData,
    string controllerName)
{
    //省略实现
}
}

```

如上面的代码片段所示, `ReflectedControllerFactory` 具有一个静态的 `controllerTypes` 字段用于保存所有被解析出来的 `Controller` 的类型。在静态构造函数中, 调用 `BuildManager` 的

GetReferencedAssemblies 方法得到所有被引用的程序集，并得到所有定义其中的实现了 IController 接口的类型，这些类型全部被添加到通过静态字段 controllerTypes 表示的类型列表。

Controller 类型的解析实现在受保护的 GetControllerType 方法中。在用于最终激活 Controller 对象的 CreateController 方法中，通过调用该方法得到与指定 RequestContext 和 Controller 名称相匹配的 Controller 类型，最终通过调用 Activator 的静态方法 CreateInstance 创建相应的 Controller 对象。

ReflectedControllerFactory 中定义了两个辅助方法，其中 IsNamespaceMatch 用于判断 Controller 类型真正的命名空间是否与指定的命名空间（可能包含统配符）相匹配，进行字符比较是忽略大小写的。私有方法 GetControllerType 根据指定的命名空间列表和类型名称匹配的类型数组得到一个完全匹配的 Controller 类型。如果得到多个匹配的类型，直接抛出 InvalidOperationException 异常，并提示具有多个匹配的 Controller 类型，如果找不到匹配类型，则返回 Null。

在如下所示的用于解析 Controller 类型的 GetControllerType 方法中，从预先得到的所有 Controller 类型列表中筛选出类型名称与传入的 Controller 名称相匹配的类型。首先通过路由对象的命名空间对之前得到的类型列表进行进一步筛选，如果能够找到一个唯一的类型，则直接将其作为 Controller 的类型返回。为了确定是否采用后备命名空间对 Controller 类型进行解析，可以从作为参数的 RouteData 对象中得到其 DataTokens 属性，并从中获取路由变量 UseNamespaceFallback 的值。如果该路由变量存在并且值为 False，则直接返回 Null。

```
public class ReflectedControllerFactory : IControllerFactory
{
    //其他成员
    protected virtual Type GetControllerType (RouteData routeData,
        string controllerName)
    {
        //根据类型名称筛选
        var types = controllerTypes.Where(type => string.Compare(
            controllerName + "Controller", type.Name, true) == 0).ToArray();
        if (types.Length == 0)
        {
            return null;
        }

        //通过路由对象的命名空间进行匹配
        var namespaces = routeData.DataTokens["Namespaces"] as
            IEnumerable<string>;
        namespaces = namespaces ?? new string[0];
        Type contrllerType = this.GetControllerType(namespaces, types);
        if (null != contrllerType)
        {
            return contrllerType;
        }

        //是否允许采用后备命名空间
        bool useNamespaceFallback = true;
        if (null != routeData.DataTokens["UseNamespaceFallback"])
        {

```

```

        useNamespaceFallback =
            (bool) (routeData.DataTokens["UseNamespaceFallback"]);
    }

    //如果不允许采用后备命名空间, 返回 Null
    if (!useNamespaceFallback)
    {
        return null;
    }

    //通过当前 ControllerBuilder 的默认命名空间进行匹配
    contrllrType = this.GetControllerType(
        ControllerBuilder.Current.DefaultNamespaces, types);
    if (null != contrllrType)
    {
        return contrllrType;
    }

    //如果只存在一个类型名称匹配的 Controller, 则返回之
    if (types.Length == 1)
    {
        return types[0];
    }

    //如果具有多个类型名称匹配的 Controller, 则抛出异常
    throw new InvalidOperationException("具有多个匹配的 Controller 类型");
}
}

```

如果 RouteData 的 DataTokens 中不存在这样一个 UseNamespaceFallback 路由变量, 或者它的值为 True, 则先采用当前 ControllerBuilder 的默认命名空间列表进一步对 Controller 类型进行解析, 如果存在唯一的类型则直接当作目标 Controller 类型返回。如果通过两组命名空间均不能得到一个匹配的 ControllerType, 并且只存在唯一一个与传入的 Controller 名称相匹配的类型, 则直接将该类型作为目标 Controller 返回。如果这样的类型具有多个, 则直接抛出 InvalidOperationException 异常。

3.2.2 Controller 类型的缓存

为了避免频繁地遍历所有程序集对目标 Controller 类型进行解析, ASP.NET MVC 对解析出来的 Controller 类型进行了缓存以提升性能。与针对用于 Area 注册的 AreaRegistration 类型的缓存类似, Controller 激活系统同样采用基于文件的缓存策略, 用于保存 Controller 类型列表的名为 MVCControllerTypeCache.xml 的文件保存在 ASP.NET 的临时目录下面。具体的路径如下:

- %Windir%\Microsoft.NET\Framework\v{version}\TemporaryASP.NET Files\{appname}\...\UserCache\
- %Windir%\Microsoft.NET\Framework\v{version}\Temporary ASP.NET Files\root\...\UserCache\

其中第一个针对寄宿于 IIS 中的 Web 应用,后者针对直接通过 Visual Studio Developer Server 作为宿主的应用。而用于保存所有 AreaRegistration 类型列表的 MVC-AreaRegistrationTypeCache.xml 文件也保存在这个目录下面。

当接收到 Web 应用被启动后的第一个请求时, Controller 激活系统会读取这个用于缓存所有 Controller 类型列表的 ControllerTypeCache.xml 文件并反序列化成一个 List<Type> 对象。只有在该列表为空的时候才会通过遍历程序集和反射的方式得到所有实现了接口 IController 的类型,而被解析出来的 Controller 类型重新被写入这个缓存文件中。这个通过读取缓存文件或者重新解析出来的 Controller 类型列表被保存到内存中,在 Web 应用活动期间内被 Controller 激活系统使用。

下面的 XML 片段反映了这个用于 Controller 类型列表缓存的 MVC-ControllerTypeCache.xml 文件的结构,从中可以看出它包含了所有的 Controller 类型的全名和所在的程序集和托管模块的名称。

```
<?xml version="1.0" encoding="utf-8"?>
<!--This file is automatically generated. Please do not modify the contents of
this file.-->
<typeCache lastModified="3/22/2012 1:18:49 PM"
    mvcVersionId="80365b23-7a1d-42b2-9e7d-cc6f5694c6d1">
  <assembly name="Artech.Admin, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null">
    <module versionId="eb343e3f-2d63-4665-a12a-29fb30dceeed">
      <type>Artech.Admin.HomeController</type>
      <type>Artech.Admin.EmployeeController </type>
    </module>
  </assembly>
  <assembly name="Artech.Portal, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null">
    <module versionId=" 3717F116-35EE-425F-A1AE-EB4267497D8C">
      <type>Artech.Portal.Controllers.HomeController</type>
      <type>Artech.Portal.ProductsController</type>
    </module>
  </assembly>
</typeCache>
```

3.2.3 Controller 的释放和会话状态行为的控制

作为激活 Controller 对象的 ControllerFactory 不仅仅用于创建目标 Controller 对象,还具有两个额外的功能,即通过 ReleaseController 方法对激活的 Controller 对象进行释放和回收,以及通过调用 GetControllerSessionBehavior 方法返回用于控制当前会话状态行为的 SessionStateBehavior 枚举对象。

对于默认使用的 DefaultControllerFactory 来说,它对 Controller 对象的释放操作很简单,即如果 Controller 类型实现了 IDisposable 接口,则直接调用其 Dispose 方法即可。我们将这个逻辑也实现在了我们自定义的 ReflectedControllerFactory 中。

```

public class ReflectedControllerFactory : IControllerFactory
{
    //其他操作
    public void ReleaseController(IController controller)
    {
        IDisposable disposable = controller as IDisposable;
        if (null != disposable)
        {
            disposable.Dispose();
        }
    }
}

```

至于用于返回 `SessionStateBehavior` 枚举的 `GetControllerSessionBehavior` 方法，在默认的情况下它的返回值为 `SessionStateBehavior.Default`。通过前面的介绍我们知道在这种情况下具体的会话状态行为取决于创建的 `HttpHandler` 所实现的标记接口。对于 ASP.NET MVC 应用来说，默认使用的 `HttpHandler` 是一个 `MvcHandler` 的对象，如下面的代码片段所示，它实现了 `IRequiresSessionState` 接口，意味着默认情况下会话状态是可读写的（相当于 `SessionStateBehavior.Required`）。

```

public class MvcHandler :
    IHttpAsyncHandler,
    IHttpHandler,
    IRequiresSessionState
{
    //其他成员
}

```

可以通过在 `Controller` 类型上应用 `System.Web.Mvc.SessionStateAttribute` 特性来具体控制会话状态行为。如下面的代码片段所示，`SessionStateAttribute` 具有一个 `SessionStateBehavior` 类型的只读属性 `Behavior` 用于返回具体行为设置的会话状态行为选项，该属性是在构造函数中被初始化的。

```

[AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = true)]
public sealed class SessionStateAttribute : Attribute
{
    public SessionStateAttribute(SessionStateBehavior behavior);
    public SessionStateBehavior Behavior { get; }
}

```

`DefaultControllerFactory` 会试着获取应用在 `Controller` 类型上的 `SessionStateAttribute` 特性，如果这样的特性存在则直接返回它的 `Behavior` 属性所表示的 `SessionStateBehavior` 枚举，如果不存在则返回 `SessionStateBehavior.Default`，具体的逻辑也反映在我们自定义的 `ReflectedControllerFactory` 的 `GetControllerSessionBehavior` 方法中。

```

public class ReflectedControllerFactory : IControllerFactory
{
    //其他成员
    public SessionStateBehavior GetControllerSessionBehavior(
        RequestContext requestContext, string controllerName)
    {

```

```

{
    Type controllerType = this.GetControllerType(requestContext.RouteData,
        controllerName);
    if (null == controllerType)
    {
        return SessionStateBehavior.Default;
    }
    SessionStateAttribute attribute = controllerType
        .GetCustomAttributes(true).OfType<SessionStateAttribute>()
        .FirstOrDefault();
    attribute = attribute ??
        new SessionStateAttribute(SessionStateBehavior.Default);
    return attribute.Behavior;
}
}

```

3.3 IoC 的应用

所谓控制反转（Inversion of Control, IoC），简单地说，就是应用本身不负责依赖对象的创建和维护，而交给一个外部容器来负责。这样控制权就由应用转移到了外部 IoC 容器，控制权就实现了所谓的反转。比如在类型 A 中需要使用类型 B 的实例，而 B 实例的创建并不由 A 来负责，而是通过外部容器来创建。通过 IoC 的方式实现针对目标 Controller 的激活具有重要的意义。

3.3.1 从 Unity 来认识 IoC

有时又将 IoC 称为依赖注入（Dependency Injection, DI）。所谓依赖注入，就是由外部容器在运行时动态地将依赖的对象注入到组件之中。Martin Fowler 在那篇著名的文章 *Inversion of Control Containers and the Dependency Injection pattern* 中将具体的依赖注入划分为三种形式，即构造器注入、属性（设置）注入和接口注入，而我个人习惯将其划分为一种（类型）匹配和三种注入。

- 类型匹配（Type Mapping）：虽然我们通过接口（或者抽象类）来进行服务调用，但是服务本身还是实现在某个具体的服务类型中，这就需要某个类型注册机制来解决服务接口和服务类型之间的匹配关系。
- 构造器注入（Constructor Injection）：IoC 容器会智能地选择和调用适合的构造函数以创建依赖的对象。如果被选择的构造函数具有相应的参数，IoC 容器在调用构造函数之前解析注册的依赖关系并自行获得相应参数对象。
- 属性注入（Property Injection）：如果需要使用到被依赖对象的某个属性，在被依赖对象被创建之后，IoC 容器会自动初始化该属性。
- 方法注入（Method Injection）：如果被依赖对象需要调用某个方法进行相应的初始化，在该对象创建之后，IoC 容器会自动调用该方法。

开源社区具有很有流行的 IoC 框架, 如 Castle Windsor、Unity、Spring.NET、StructureMap 和 Ninject 等。Unity 是微软 Patterns& Practices 部门开发的一个轻量级的 IoC 框架, 该项目在 Codeplex 上的地址为 <http://unity.codeplex.com/>, 我们可以下载相应的安装包和开发文档。在本书出版之时, Unity 的最新版本为 2.1。出于篇幅的限制, 我们不可能对 Unity 进行详细的讨论, 但是为了让读者了解 IoC 在 Unity 中的实现, 我们写了一个简单的程序。

创建一个控制台程序, 定义如下几个接口 (IA、IB、IC 和 ID) 和它们各自的实现类 (A、B、C、D)。在类型 A 中定义了 B、C 和 D 3 个属性, 其类型分别为接口 IB、IC 和 ID。属性 B 在函数中被初始化, 意味着它会以构造器注入的方式被初始化; 属性 C 上应用了 `Microsoft.Practices.Unity.DependencyAttribute` 特性, 意味着这是一个需要以属性注入方式被初始化的依赖属性; 属性 D 则通过方法 `Initialize` 初始化, 该方法上应用了特性 `Microsoft.Practices.Unity.InjectionMethodAttribute`, 意味着这是一个注入方法, 它会在 A 对象被 IoC 容器创建的时候会被自动调用。

```
namespace UnityDemo
{
    public interface IA { }
    public interface IB { }
    public interface IC { }
    public interface ID {}

    public class A : IA
    {
        public IB B { get; set; }
        [Dependency]
        public IC C { get; set; }
        public ID D { get; set; }

        public A(IB b)
        {
            this.B = b;
        }
        [InjectionMethod]
        public void Initialize(ID d)
        {
            this.D = d;
        }
    }
    public class B: IB{}
    public class C: IC{}
    public class D: ID{}
}
```

然后为该应用添加一个配置文件, 并定义如下一段关于 Unity 的配置。这段配置定义了一个名称为 `defaultContainer` 的 Unity 容器, 并在其中完成了上面定义的接口和对应实现类之间映射的类型匹配。

```
<configuration>
  <configSections>
    <section name="unity"
      type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
```



```

        Microsoft.Practices.Unity.Configuration"/>
    </configSections>
    <unity>
        <containers>
            <container name="defaultContainer">
                <register type="UnityDemo.IA, UnityDemo" mapTo="UnityDemo.A, UnityDemo"/>
                <register type="UnityDemo.IB, UnityDemo" mapTo="UnityDemo.B, UnityDemo"/>
                <register type="UnityDemo.IC, UnityDemo" mapTo="UnityDemo.C, UnityDemo"/>
                <register type="UnityDemo.ID, UnityDemo" mapTo="UnityDemo.D, UnityDemo"/>
            </container>
        </containers>
    </unity>
</configuration>

```

最后在作为程序入口的 Main 方法中创建一个代表 IoC 容器的 UnityContainer 对象，并加载配置信息对其进行初始化。然后调用它的泛型方法 Resolve 创建一个实现了泛型接口 IA 的对象。最后将返回对象转变成类型 A，并检验其 B、C 和 D 属性是否为 Null。

```

static void Main(string[] args)
{
    IUnityContainer container = new UnityContainer();
    UnityConfigurationSection configuration =
        ConfigurationManager.GetSection(UnityConfigurationSection.SectionName)
        as UnityConfigurationSection;
    configuration.Configure(container, "defaultContainer");
    A a = container.Resolve<IA>() as A;
    if (null != a)
    {
        Console.WriteLine("a.B == null ? {0}", a.B == null ? "Yes" : "No");
        Console.WriteLine("a.C == null ? {0}", a.C == null ? "Yes" : "No");
        Console.WriteLine("a.D == null ? {0}", a.D == null ? "Yes" : "No");
    }
}

```

从如下给出的执行结果可以得到这样的结论：通过 Resolve<IA>方法返回的是一个类型为 A 的对象，该对象的三个属性被进行了有效的初始化。这个简单的程序分别体现了接口注入（通过相应的接口根据配置解析出相应的实现类型）、构造器注入（属性 B）、属性注入（属性 C）和方法注入（属性 D）。(S309)

```

a.B == null ? No
a.C == null ? No
a.D == null ? No

```

3.3.2 Controller 与 Model 的分离

在第 1 章“ASP.NET + MVC”中我们谈到过 ASP.NET MVC 是基于 MVC 的变体 Model2 设计的。ASP.NET MVC 所谓的 Model 仅仅表示绑定到 View 上的数据，我们一般称之为 View Model。而真正的 Model 一般意义上指维护应用状态和提供业务功能操作的领域模型，或者是针对业务层的入口或者业务服务的代理。真正的 MVC 在 ASP.NET MVC 中的体现如图 3-8 所示。

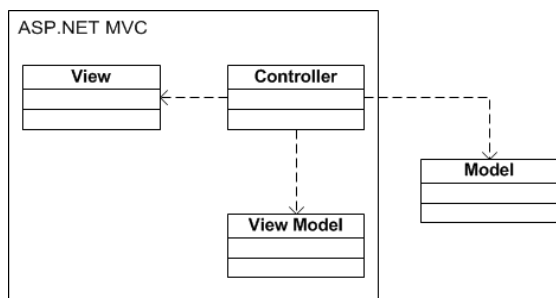


图 3-8 ASP.NET MVC + Model

对于一个 ASP.NET MVC 应用来说，用户交互请求直接发送给 Controller。如果涉及针对某项业务功能的调用，Controller 会直接调用 Model。如果需要呈现业务数据，Controller 会通过 Model 获取相应业务数据并转换成 View Model，最终通过 View 呈现出来，这样的交互协议方式反映了 Controller 针对 Model 的直接依赖。

如果我们在 Controller 激活系统中引入 IoC，并采用 IoC 的方式提供用于处理请求的 Controller 对象，那么 Controller 和 Model 之间的依赖程度在很大程度上被降低了，甚至可以像图 3-9 所示的一样，以接口的方式对 Model 进行抽象，让 Controller 依赖于这个抽象化的 Model 接口，而不是具体的 Model 实现。

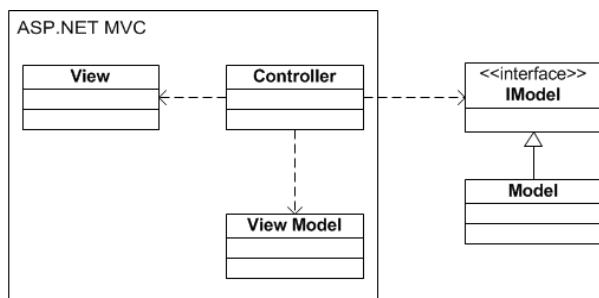


图 3-9 ASP.NET MVC + IModel + Model

3.3.3 基于 IoC 的 ControllerFactory

ASP.NET MVC 的 Controller 激活系统最终通过 ControllerFactory 来创建目标 Controller 对象，要将 IoC 引入 ASP.NET MVC 并通过对应的 IoC 容器实现对目标 Controller 的激活，我们很自然地会想到自定义一个基于 IoC 的 ControllerFactory。

对于自定义 ControllerFactory，可以直接实现 IControllerFactory 接口创建一个全新的 ControllerFactory 类型，这需要实现包括 Controller 类型的解析、Controller 实例的创建与释放以及会话状态行为选项的获取在内的所有功能。一般来说，Controller 实例的创建才需要 IoC 容器的控制，为了避免重新实现其他的功能，可以直接继承 DefaultControllerFactory，重写 Controller 实例创建的逻辑。

实例演示：创建基于 Unity 的 ControllerFactory (S310)

现在我们通过一个简单的实例演示如何通过自定义 ControllerFactory 利用 Unity 进行 Controller 的激活。为了避免针对 Controller 类型解析、会话状态行为选项的获取和对 Controller 对象的释放逻辑的重复定义，我们直接继承 DefaultControllerFactory。将该自定义 ControllerFactory 命名为 UnityControllerFactory。如下面的代码片段所示，UnityControllerFactory 仅仅重写了受保护的虚方法 GetControllerInstance，将成功解析的 Controller 类型作为调用 UnityContainer 的 Resolve 方法的参数，而返回值就是需要被激活的 Controller 实例。

```
public class UnityControllerFactory: DefaultControllerFactory
{
    public IUnityContainer UnityContainer { get; private set; }
    public UnityControllerFactory(IUnityContainer unityContainer)
    {
        this.UnityContainer = unityContainer;
    }
    protected override IController GetControllerInstance(
        RequestContext requestContext, Type controllerType)
    {
        if (null == controllerType)
        {
            return null;
        }
        return (IController)this.UnityContainer.Resolve(controllerType);
    }
}
```

整个自定义的 UnityControllerFactory 就这么简单。为了演示 IoC 在它身上的体现，我们 在一个简单的 ASPMVC 实例中来使用它。我们沿用在第 2 章“URL 路由”中使用过的关于 “员工管理”的场景，如图 3-10 所示，本实例由两个页面（对应着两个 View）组成，一个 用于显示员工列表，另一个用于显示基于某个员工的详细信息。

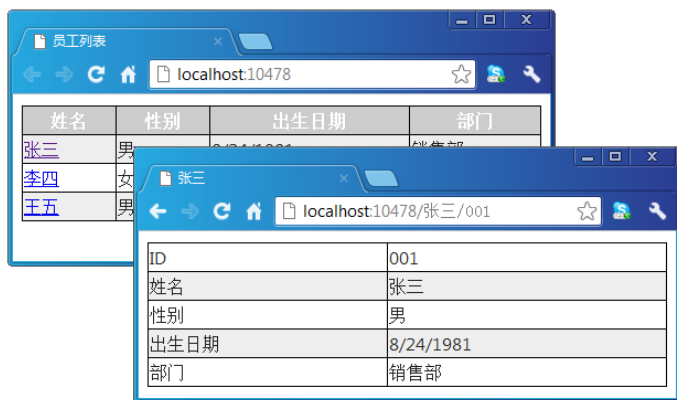


图 3-10 员工列表和员工详细信息页面

在一个 ASP.NET MVC 应用中添加对 Unity 的程序集 Microsoft.Practices.Unity.dll 的引用（如果读者不想安装 Unity，可以通过下载本实例的源代码的方式获取该程序集），然后在

Models 目录下定义如下一个表示员工信息的 Employee 类型。

```
public class Employee
{
    [Display(Name="ID")]
    public string Id { get; private set; }

    [Display(Name = "姓名")]
    public string Name { get; private set; }

    [Display(Name = "性别")]
    public string Gender { get; private set; }

    [Display(Name = "出生日期")]
    [DataType(DataType.Date)]
    public DateTime BirthDate { get; private set; }

    [Display(Name = "部门")]
    public string Department { get; private set; }

    public Employee(string id, string name, string gender, DateTime birthDate,
        string department)
    {
        this.Id = id;
        this.Name = name;
        this.Gender = gender;
        this.BirthDate = birthDate;
        this.Department = department;
    }
}
```

我们创建一个 `EmployeeRepository` 对象来进行数据的获取，并为它定义了对应的接口 `IEmployeeRepository`。如下面的代码片段所示，`IEmployeeRepository` 仅仅具有一个返回 `Employee` 列表的唯一方法 `GetEmployees`，用于获取指定 ID 的员工信息。如果指定的 ID 为空，则返回所有员工列表。`EmployeeRepository` 直接利用一个静态字段模拟对数据的存储。

```
public interface IEmployeeRepository
{
    IEnumerable<Employee> GetEmployees(string id = "");
}

public class EmployeeRepository : IEmployeeRepository
{
    private static IList<Employee> employees;

    static EmployeeRepository()
    {
        employees = new List<Employee>();
        employees.Add(new Employee("001", "张三", "男", new DateTime(1981, 8, 24),
            "销售部"));
        employees.Add(new Employee("002", "李四", "女", new DateTime(1982, 7, 10),
            "人事部"));
        employees.Add(new Employee("003", "王五", "男", new DateTime(1981, 9, 21),
            "人事部"));
    }
}
```

```

public IEnumerable<Employee> GetEmployees(string id = "")
{
    return employees.Where(e => e.Id == id || string.IsNullOrEmpty(id));
}

```

我们创建了一个具有如下定义的 `EmployeeController`，它具有一个类型为 `IEmployeeRepository` 的属性 `Repository`，应用在上面的 `DependencyAttribute` 特性告诉我们这是一个“依赖属性”。当我们采用 `UnityContainer` 来激活 `EmployeeController` 对象的时候，会根据注册的类型映射来实例化一个实现了 `IEmployeeRepository` 的类型的实例来初始化该属性。

```

public class EmployeeController : Controller
{
    [Dependency]
    public IEmployeeRepository Repository { get; set; }

    public ActionResult GetAllEmployees()
    {
        var employees = this.Repository.GetEmployees();
        return View("EmployeeList", employees);
    }

    public ActionResult GetEmployeeById(string id)
    {
        Employee employee = this.Repository.GetEmployees(id).FirstOrDefault();
        if (null == employee)
        {
            throw new HttpException(404, string.Format("ID 为{0}的员工不存在", id));
        }
        return View("Employee", employee);
    }
}

```

`EmployeeController` 定义了两个基本的 Action 方法。`GetAllEmployees` 通过 `Repository` 获取所有员工列表并将其通过名为 `EmployeeList` 的 View 呈现出来。另一个 Action 方法 `GetEmployeeById` 根据指定的 ID 获取相应的员工信息，最终用于呈现单个员工信息的 View 为 `Employee`。如果根据指定的 ID 找不到相应的员工，直接抛出一个状态为“404”的 `HttpException` 异常。

如下所示的是用于显示员工列表的 View（`EmployeeList`）的定义，它的 Model 类型为 `IEnumerable<Employee>`。在该 View 中，通过一个表格来显示员工列表，值得一提的是，可以通过调用 `HtmlHelper` 的 `ActionLink` 方法将员工的名称显示为一个指向 Action 方法 `GetEmployeeById` 的链接。

```

@model IEnumerable<Employee>
<html>
    <head>
        <title>员工列表</title>
    </head>
    <body>

```

```

<table>
  <tr>
    <th>姓名</th>
    <th>性别</th>
    <th>出生日期</th>
    <th>部门</th>
  </tr>
  @{
    foreach(Employee employee in Model)
    {
      <tr>
        <td>@Html.ActionLink(employee.Name, "GetEmployeeById",
          new { name = employee.Name, id = employee.Id })
        </td>
        <td>@Html.DisplayFor(m=>employee.Gender)</td>
        <td>@Html.DisplayFor(m=>employee.BirthDate)</td>
        <td>@Html.DisplayFor(m=>employee.Department)</td>
      </tr>
    }
  }
</table>
</body>
</html>

```

用于显示单个员工信息的名为 `Employee` 的 `View` 定义如下，这是一个 `Model` 类型为 `Employee` 的强类型的 `View`，通过表格的形式将员工的详细信息显示出来。

```

@model Employee
<html>
  <head>
    <title>@Model.Name</title>
  </head>
  <body>
    <table>
      <tr>
        <td>
          @Html.LabelFor(m=>m.Id)</td><td>@Html.DisplayFor(m=>m.Id)
        </td>
      </tr>
      <tr>
        <td>
          @Html.LabelFor(m=>m.Name)</td><td>@Html.DisplayFor(
            m=>m.Name)
          </td>
        </tr>
      <tr>
        <td>
          @Html.LabelFor(m=>m.Gender)</td><td>@Html.DisplayFor(
            m=>m.Gender)
          </td>
        </tr>
      <tr>
        <td>
          @Html.LabelFor(m=>m.BirthDate)</td><td>@Html.DisplayFor(
            m=>m.BirthDate)
          </td>

```

```

        </tr>
        <tr>
            <td>
                @Html.LabelFor(m=>m.Department)</td><td>@Html.DisplayFor(
                    m=>m.Department)
            </td>
        </tr>
    </table>
</body>
</html>

```

我们对两个页面的 URL 进行了相应的设计，主页用于显示所有员工列表，它指向 `EmployeeController` 的 Action 方法 `GetAllEmployees`。用于显示单个员工详细信息的页面的 URL 的结构为 “/{员工姓名}/{员工 ID}”（比如 “/李四/002”），它自然指向另一个 Action 方法 `GetEmployeeById`，为此我们在自动生成的 `RouteConfig` 类型中按照如下的方式注册两个路由。

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        //其他操作
        routes.MapRoute(
            name: "Home",
            url: "",
            defaults: new { controller = "Employee", action = "GetAllEmployees" }
        );

        routes.MapRoute(
            name: "Detail",
            url: "{name}/{id}",
            defaults: new { controller = "Employee", action = "GetEmployeeById" }
        );
    }
}

```

自定义的 `ControllerFactory` (`UnityControllerFactory`) 在 `Global.asax` 中通过如下的代码进行注册。用于创建 `UnityControllerFactory` 的 `UnityContainer` 对象注册了 `IEmployeeRepository` 和 `EmployeeRepository` 之间的映射关系。

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        UnityContainer unityContainer = new UnityContainer();
        unityContainer.RegisterType<IEmployeeRepository, EmployeeRepository>();
        UnityControllerFactory controllerFactory =
            new UnityControllerFactory(unityContainer);
        ControllerBuilder.Current.SetControllerFactory(controllerFactory);
    }
}

```

除此之外，我们还为该实例应用定义相应的布局文件和 CSS 样式，在这里就不一一介

绍了。这个例子旨在演示通过自定义 `ControllerFactory` 实现以 IoC 的方式激活目标 `Controller` 对象，这样可以最大限度地降低 `Controller` 和其他组件之间的依赖关系，因为这些依赖会被用于激活 `Controller` 的 IoC 容器动态注入。

3.3.4 基于 IoC 的 `ControllerActivator`

除了通过自定义 `ControllerFactory` 的方式引入 IoC 之外，在使用默认 `DefaultControllerFactory` 情况下也可以通过一些扩展使基于 IoC 的 `Controller` 激活成为可能。不过这就需要我们具体了解实现在 `DefaultControllerFactory` 内部的 `Controller` 激活机制了。

`DefaultControllerFactory` 针对目标 `Controller` 的激活其实是通过另一个名为 `ControllerActivator` 的组件来完成的，所有的 `ControllerActivator` 实现了 `System.Web.Mvc.IControllerActivator` 接口。如下面的代码片段所示，`IControllerActivator` 定义了唯一的用于创建 `Controller` 对象的 `Create` 方法，而 `DefaultControllerFactory` 使用的 `ControllerActivator` 可以直接通过构造函数参数的方式来指定。

```
public interface IControllerActivator
{
    IController Create(RequestContext requestContext, Type controllerType);
}

public class DefaultControllerFactory : IControllerFactory
{
    //其他成员
    public DefaultControllerFactory();
    public DefaultControllerFactory(IControllerActivator controllerActivator);
}
```

实例演示：创建基于 Ninject 的 `ControllerActivator` (S311)

如果我们基于一个 `ControllerActivator` 对象来创建一个 `DefaultControllerFactory`，它最终会被用于 `Controller` 对象的激活，那么可以通过自定义 `ControllerActivator` 的方式将 IoC 引入 `Controller` 激活系统。接来自定义的 `ControllerActivator` 基于另一个 IoC 框架 `Ninject`，较之 `Unity`，`Ninject` 是一个更加轻量级也更适合 `ASP.NET MVC` 的 IoC 框架，将自定义的 `ControllerActivator` 起名为 `NinjectControllerActivator`。如下面的代码所示，针对目标 `Controller` 的创建是通过一个 `StandardKernel` 对象来完成的，为了方便实现类型的映射，我们定义了一个泛型的 `Register` 方法。

```
public class NinjectControllerActivator : IControllerActivator
{
    public IKernel Kernel { get; private set; }

    public NinjectControllerActivator()
    {
        this.Kernel = new StandardKernel();
    }
}
```



```

    }

    public IController Create(RequestContext requestContext, Type controllerType)
    {
        return (IController)this.Kernel.TryGet(controllerType);
    }

    public void Register<TFrom, TTo>() where TTo: TFrom
    {
        this.Kernel.Bind<TFrom>().To<TTo>();
    }
}

```

接下来我们使用的还是之前演示过的关于员工管理的例子，前面我们演示了属性注入的方式在激活 `EmployeeController` 的时候对 `Repository` 进行初始化，现在来演示另一种依赖注入形式——构造器注入。如下面的代码片段所示，只读的 `Repository` 是在构造函数中通过指定的参数初始化的，而该参数的类型是 `IEmployeeRepository`。

```

public class EmployeeController : Controller
{
    //其他成员
    public IEmployeeRepository Repository { get; private set; }

    public EmployeeController(IEmployeeRepository repository)
    {
        this.Repository = repository;
    }
}

```

为了让 ASP.NET MVC 的 Controller 激活系统采用我们自定义的 `ControllerActivator` 来创建目标 Controller，我们需要创建并注册一个相应的 `DefaultControllerFactory` 对象。如下面的代码片段所示，我们在 `Global.asax` 中创建一个 `NinjectControllerActivator` 对象，并注册了接口 `IEmployeeRepository` 和实现类型 `EmployeeRepository` 之间的匹配关系。最后据此创建一个 `DefaultControllerFactory` 对象，通过当前的 `ControllerBuilder` 进行注册。

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他成员
        NinjectControllerActivator controllerActivator =
            new NinjectControllerActivator();
        controllerActivator.Register<IEmployeeRepository, EmployeeRepository>();
        DefaultControllerFactory controllerFactory =
            new DefaultControllerFactory(controllerActivator);
        ControllerBuilder.Current.SetControllerFactory(controllerFactory);
    }
}

```

再次运行我们的程序，依然会得到如图 3-10 所示的结果，其实自定义 `ControllerActivator` 实现 IoC 的方式并不是很常用，接下来我们介绍第三种更加常用的 IoC 实现方式。

3.3.5 基于 IoC 的 DependencyResolver

如果在构建 `DefaultControllerFactory` 的时候没有显式指定采用 `ControllerActivator`，它默认使用的是一个类型为 `DefaultControllerActivator` 的对象。如下面的代码片段所示，这只是一个实现了 `IControllerActivator` 接口的私有类型，不能直接通过编程的方式使用它。

```
private class DefaultControllerActivator : IControllerActivator
{
    public DefaultControllerActivator();
    public DefaultControllerActivator(IDependencyResolver resolver);
    public IController Create(RequestContext requestContext,
        Type controllerType);
}
```

即使 `DefaultControllerFactory` 采用了默认的 `DefaultControllerActivator`，依然可以将 IoC 引入到 `Controller` 的激活系统中，而这就需要进一步了解实现在 `DefaultControllerActivator` 的 `Controller` 激活逻辑了。

其实 `DefaultControllerActivator` 完成对 `Controller` 的激活依赖于另一个名为 `DependencyResolver` 的对象。`DependencyResolver` 是一个非常重要的组件，可以将其视为 ASP.NET MVC 框架内部使用的 IoC 容器。它不只是用于针对 `Controller` 的激活，框架内部很多组件的提供最终都依赖于它。`DependencyResolver` 实现了具有如下定义的 `System.Web.Mvc.IDependencyResolver` 接口，`GetService` 和 `GetServices` 方法分别用于根据指定的类型获取单个和所有实例。

```
public interface IDependencyResolver
{
    object GetService(Type serviceType);
    IEnumerable<object> GetServices(Type serviceType);
}
```

整个 Web 默认使用的 `DependencyResolver` 可以通过 `System.Web.Mvc.DependencyResolver` 类型进行注册。如下面的代码片段所示，`DependencyResolver` 类型具有一个静态的 `Current` 属性表示当前 `DependencyResolver`，具体对 `DependencyResolver` 的注册通过调用静态方法 `SetResolver` 来完成。顺便说一下，`DependencyResolver` 类型并没有实现 `IDependencyResolver` 接口，并不是真正意义上的 `DependencyResolver`。

```
public class DependencyResolver
{
    //其他成员
    private static DependencyResolver _instance;

    public void InnerSetResolver(object commonServiceLocator);
    public void InnerSetResolver(IDependencyResolver resolver);
    public void InnerSetResolver(Func<Type, object> getService,
        Func<Type, IEnumerable<object>> getServices);

    public static void SetResolver(object commonServiceLocator);
    public static void SetResolver(IDependencyResolver resolver);
}
```

```

    public static void SetResolver(Func<Type, object> getService,
        Func<Type, IEnumerable<object>> getServices);

    public static IDependencyResolver Current { get; }
    public IDependencyResolver InnerCurrent { get; }
}

```

这个被封装的 `DependencyResolver` (指实现了接口 `IDependencyResolver` 的某个类型的对象, 不是指 `DependencyResolver` 类型的对象, 对于后者我们会采用“`DependencyResolver` 类型对象”的说法) 通过只读属性 `InnerCurrent` 表示, 而三个 `InnerSetResolver` 方法重载用于初始化该属性。静态字段 `_instance` 表示当前的 `DependencyResolver` 类型对象, 静态只读属性 `Current` 则表示该对象内部封装的 `DependencyResolver` 对象, 而它通过三个静态的 `SetResolver` 进行初始化。

如果没有对 `DependencyResolver` 进行显式注册, 系统默认使用的是一个类型为 `DefaultDependencyResolver` 的对象。如下面的代码片段所示, 这是一个私有类型, 用于根据类型提供“服务实例”的 `GetService` 方法直接以反射的方式根据类型创建并返回对应的实例。对于类型为接口/抽象类, 或者不曾定义默认公有构造函数的类型, 我们直接返回 `Null`。也就是说在默认的情况下, `Controller` 的激活最终是通过对 `Controller` 类型的反射来实现的。`DefaultDependencyResolver` 的另一个 `GetServices` 方法直接返回一个空的对象列表。

```

private class DefaultDependencyResolver : IDependencyResolver
{
    public object GetService(Type serviceType)
    {
        if (serviceType.IsInterface || serviceType.IsAbstract)
        {
            return null;
        }
        try
        {
            return Activator.CreateInstance(serviceType);
        }
        catch
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return Enumerable.Empty<object>();
    }
}

```

上面介绍的类型 `DefaultControllerFactory`、`IControllerActivator`、`DefaultControllerActivator`、`IDependencyResolver`、`DefaultDependencyResolver` 和 `DependencyResolver` 之前的关系基本上可以通过如图 3-11 所示的类图来体现。

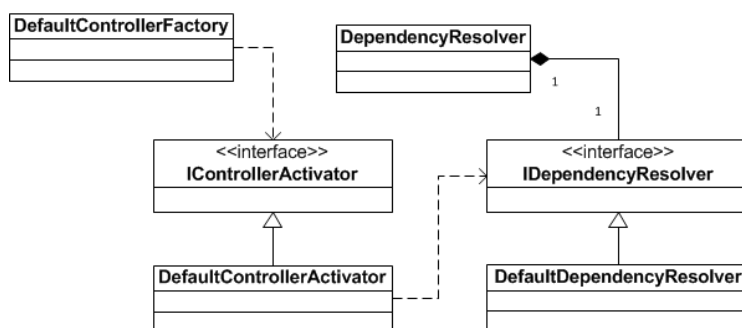


图 3-11 DefaultControllerFactory + ControllerActivator + DependencyResolver

实例演示：创建基于 Ninject 的 DependencyResolver (S312)

通过前面的介绍我们知道，当调用构造函数创建一个 DefaultControllerFactory 的时候，如果调用的时候默认无参数构造函数，后者将作为参数的 ControllerActivator 对象设置为 Null，那么默认请求用于激活 Controller 实例的是通过 DependencyResolver 类型的静态属性 Current 表示的 DependencyResolver 对象，换言之，我们可以通过自定义 DependencyResolver 的方式来实现基于 IoC 的 Controller 激活。

同样是采用 Ninject，我们定义了一个具有如下定义的 NinjectDependencyResolver。与上面定义的 NinjectControllerActivator 类似，NinjectDependencyResolver 具有一个 IKernal 类型的只读属性 Kernel，该属性在构造函数中被初始化为一个 StandardKernel 对象。对于实现的 GetService 和 GetServices 方法，直接调用 Kernel 的 TryGet 和 GetAll 返回指定类型的实例和实例列表。为了方便进行类型映射，我们定义了泛型的 Register<TFrom,TTo> 方法。

```

public class NinjectDependencyResolver : IDependencyResolver
{
    public IKernal Kernel { get; private set; }

    public NinjectDependencyResolver()
    {
        this.Kernel = new StandardKernel();
    }

    public void Register<TFrom, TTo>() where TTo: TFrom
    {
        this.Kernel.Bind<TFrom>().To<TTo>();
    }

    public object GetService(Type serviceType)
    {
        return this.Kernel.TryGet(serviceType);
    }
}
  
```

```

public IEnumerable<object> GetServices(Type serviceType)
{
    return this.Kernel.GetAll(serviceType);
}
}

```

我们只需要创建一个自定义的 `NinjectDependencyResolver` 对象并将其作为当前的 `DependencyResolver` 即可。如下面的代码片段所示，我们创建了一个 `NinjectDependencyResolver` 对象并注册了 `IEmployeeRepository` 和 `EmployeeRepository` 之间的映射关系，然后调用 `DependencyResolver` 的静态方法 `SetResolver` 将创建的 `NinjectDependencyResolver` 注册为当前的 `DependencyResolver` 对象。再次运行我们的程序，依然会得到如图 3-10 所示的效果。

```

public class MvcApplication : System.Web.HttpApplication
{
    //其他成员
    protected void Application_Start()
    {
        //其他操作
        NinjectDependencyResolver dependencyResolver =
            new NinjectDependencyResolver();
        dependencyResolver.Register<IEmployeeRepository, EmployeeRepository>();
        DependencyResolver.SetResolver(dependencyResolver);
    }
}

```

本章小结

当目标 `Controller` 的名称通过 URL 路由被解析出来之后，ASP.NET MVC 利用注册的 `ControllerFactory` 根据该名称实现对目标 `Controller` 的激活。除了完成对 `Controller` 的激活之外，`ControllerFactory` 还负责对 `Controller` 的释放工作，以及获取用于控制会话状态行为的 `SessionStateBehavior` 枚举。`ControllerFactory` 的注册通过 `ControllerBuilder` 来完成。

ASP.NET MVC 默认使用的 `ControllerFactory` 类型为 `DefaultControllerFactory`，它在对 `Controller` 类型进行解析的时候对所有 `Controller` 类型采用了基于文件的缓存以提升性能。在 `DefaultControllerFactory` 内部，它将解析得到的 `Controller` 类型递交给 `ControllerActivator` 对象对 `Controller` 实施最终的激活。默认使用 `DefaultControllerActivator` 内部利用了当前注册的 `DependencyResolver` 来提供具体的 `Controller` 对象。如果没有对 `DependencyResolver` 进行显式注册，默认提供的 `DependencyResolver` 将采用对提供类型的反射方式创建相应的实例

将 IoC 应用到 `Controller` 的激活过程中具有重要的意义，可以极大地降低 `Controller` 和其他组件的依赖关系。通过对 `Controller` 激活流程的分析，我们提供了三种实现方法，即自定义 `ControllerFactory`、`ControllerActivator` 和 `DependencyResolver`。

第 4 章 Model 元数据的解析

ASP.NET MVC 的 Model 为 View Model,表示最终呈现在 View 上的数据,而 Model 元数据的一个主要的作用在于控制 Model 对象在 View 上的呈现方式。说得更加具体点,基于某种数据类型的 Model 元数据用于指导最终生成怎样的 HTML 来呈现对应的 Model 对象。Model 元数据的存在使模板化的 HTML 呈现机制成为可能。除此之外,Model 元数据还服务于 Model 绑定和 Model 验证。

4.1 Model 元数据及其定制

Model 元数据是针对数据类型的一种描述信息，主要用于控制数据类型本身及其成员属性在界面上的呈现方式，同时也为 Model 绑定和验证提供必不可少的元数据信息。一个复杂数据类型通过属性的方式定义了一系列的数据成员，而 Model 元数据不仅仅是数据类型本身的描述，对数据成员的描述也包含其中，所以 Model 元数据具有一个层次化结构。

4.1.1 Model 元数据层次化结构

举个例子，假设我们的 Model 类型是如下一个表示联系人的 Contact 类，其属性 Name、PhoneNo、EmailAddress 和 Address 分别代表姓名、电话号码、邮箱地址和联系地址。联系地址通过另一个数据类型 Address 表示，属性 Province、City、District 和 Street 分别表示所在省份、城市、城区和街道。

```
public class Contact
{
    public string      Name { get; set; }
    public string      PhoneNo { get; set; }
    public string      EmailAddress { get; set; }
    public Address      Address { get; set; }
}

public class Address
{
    public string Province { get; set; }
    public string City { get; set; }
    public string District { get; set; }
    public string Street { get; set; }
}
```

基于 Contact 类型的 Model 元数据不仅仅具有 Contact 类型本身和其属性成员的描述，由于其 Address 属性依然是一个复杂类型，元数据还需要描述定义在该类型中的四个属性成员。图 4-1 给出了基于 Contact 类型的 Model 元数据的层次化结构。

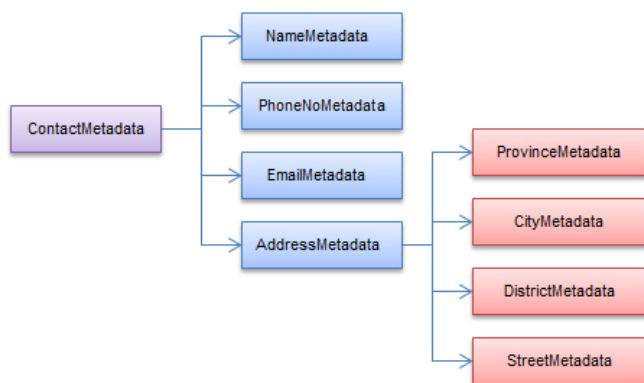


图 4-1 基于 Contact 类型的 Model 元数据的层次化结构

ASP.NET MVC 通过类型 `System.Web.Mvc.ModelMetadata` 来表示 Model 元数据，Model 元数据的层次化结构同样可以从它的定义看出来。如下面的代码片段所示，`ModelMetadata` 具有一个类型为 `ModelMetadata` 列表的只读属性 `Properties`，表示用于描述属性的 Model 元数据列表。

```
public class ModelMetadata
{
    //其他成员
    public virtual IEnumerable<ModelMetadata> Properties { get; }
}
```

由于基于类型的 `ModelMetadata` 和基于数据成员的 `ModelMetadata` 是一种包含关系，可以将前者称为后者的容器（Container）。`ModelMetadata` 的层次化结构可以通过图 4-2 所示的 UML 图来体现。

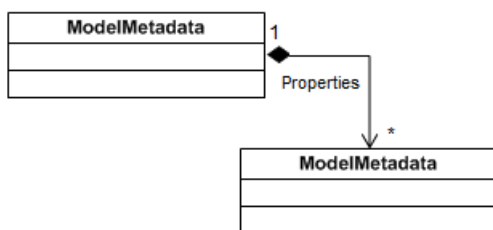


图 4-2 `ModelMetadata` 层次化结构

4.1.2 基本 Model 元数据信息

用于描述数据类型的 Model 元数据主要在 View 中为绑定的数据实现模板化的 HTML 呈现，所以很多元数据信息都与数据呈现有关。对于这些元数据信息及其如何控制 HTML，我们会在后面进行单独介绍，在这里先来看看 `ModelMetadata` 类型中与此无关的一些属性。

```
public class ModelMetadata
{
    //其他成员
    public Type           ModelType { get; }
    public virtual bool   IsComplexType { get; }
    public bool           IsNullableValueType { get; }
    public Type           ContainerType { get; }

    public object         Model { get; set; }
    public string         PropertyName { get; }

    public virtual Dictionary<string, object> AdditionalValues { get; }
    protected ModelMetadataProvider Provider { get; set; }
}
```

如上面的代码片段所示，`ModelMetadata` 具有四个与类型相关的只读属性。`ModelType`

表示 Model 本身的类型，比如说针对上面定义的 Contact 类型的 ModelMetadata 对象，其 ModelType 属性值就是 Contact 类型。而针对其属性的 ModelMetadata 对象，则具体的属性类型将作为它的 ModelType 属性。属性 IsComplexType 和 IsNullableValueType 分别表示以 ModelType 属性表示的 Model 类型是一个复杂类型和可空值类型。

Model 元数据具有树形的层次结构，某个节点可以看成其子节点的容器，而 ContainerType 则是表示容器的类型。同样以前面定义的 Contact 类型为例，基于该类型本身的 ModelMetadata 是整个层次树的根节点，所以 ContainerType 返回 Null。基于属性 Address 的 ModelMetadata 的 ContainerType 属性返回 Contact 类型，而基于 Address 的属性 Province 的 ModelMetadata 的 ContainerType 属性值则是 Address 类型。

ModelMetadata 的 Model 属性表示具体的数据对象。针对类型 Contact 的 ModelMetadata 的 Model 属性值为具体的某个 Contact 对象，而针对其中某个属性的 ModelMetadata 的 Model 属性则对应着相应的属性值。值得一提的是，Model 属性是可读可写的，可以随时根据需要改变它。另一个属性 PropertyName 表示对应的属性值，对于根节点 ModelMetadata 来说，该属性总是返回 Null。

ModelMetadata 的 AdditionalValues 属性返回一个字典对象，用于存储一些自定义的属性，字典元素的 Key 和 Value 分别代表自定义属性的名称和值。对于自定义属性的添加，可以在数据类型或者其数据成员上应用 System.Web.Mvc.AdditionalMetadataAttribute 特性来实现。如下面的代码片段所示，AdditionalMetadataAttribute 具有 Name 和 Value 两个只读属性，它们分别表示自定义属性的名称和对应的值，直接通过构造函数进行初始化。AdditionalMetadataAttribute 实现了 System.Web.Mvc.IMetadataAware 接口，对于 Model 元数据的定制来说，这是一个非常重要并且实用的接口，将在下一节对其进行单独介绍。

```
[AttributeUsage(AttributeTargets.Interface | AttributeTargets.Property |
    AttributeTargets.Class, AllowMultiple=true)]
public sealed class AdditionalMetadataAttribute : Attribute, IMetadataAware
{
    public AdditionalMetadataAttribute(string name, object value);
    public void OnMetadataCreated(ModelMetadata metadata);
    public string Name { get; }
    public object Value { get; }
}
```

ModelMetadata 的属性 Provider 是一个 System.Web.Mvc.ModelMetadataProvider 对象，从类型命名不难看出它是 ModelMetadata 的提供者。我们将在本章后续部分对以 ModelMetadataProvider 为核心的 Model 元数据提供机制进行详细介绍。

复杂类型还是简单类型？

ModelMetadata 的 IsComplexType 属性用于判断 Model 类型是简单类型还是复杂类型。

在这里判断某个类型是否是复杂类型的条件只有一个，即是否允许字符串类型向该类型的转换。所以所有的基元类型（Primitive Type）和可空值类型（Nullable Type）均不是复杂类型。对于一个默认为复杂类型的自定义数据类型，可以在它上面应用 `TypeConverterAttribute` 特性并指定一个支持字符串转换的 `TypeConverter` 类型，使之转变成简单类型。

如下面的代码片段所示，我们定义了一个表示二维坐标的 `Point` 类型，并通过应用在上方的 `TypeConverterAttribute` 特性指定了一个类型为 `PointTypeConverter` 的 `TypeConverter`。由于 `PointTypeConverter` 支持从字符串到 `Point` 类型的转换，所以 `Point` 并不是一个复杂类型。

```
[TypeConverter(typeof(PointTypeConverter))]
public class Point
{
    public double X { get; set; }
    public double Y { get; set; }

    public Point(double x, double y)
    {
        this.X = x;
        this.Y = y;
    }

    public static Point Parse(string point)
    {
        string[] split = point.Split(',');
        if(split.Length != 2)
        {
            throw new FormatException("Invalid point expression.");
        }
        double x;
        double y;
        if (!double.TryParse(split[0], out x) ||
            !double.TryParse(split[1], out y))
        {
            throw new FormatException("Invalid point expression.");
        }
        return new Point(x, y);
    }
}

public class PointTypeConverter : TypeConverter
{
    public override bool CanConvertFrom(ITypeDescriptorContext context,
        Type sourceType)
    {
        return sourceType == typeof(string);
    }

    public override object ConvertFrom(ITypeDescriptorContext context,
        CultureInfo culture, object value)
    {
        if (value is string)
        {
            return new Point(
                double.Parse(((string) value).Split(',')[0]),
                double.Parse(((string) value).Split(',')[1]));
        }
    }
}
```

```

        return Point.Parse(value as string);
    }
    return base.ConvertFrom(context, culture, value);
}
}

```

4.1.3 Model 元数据的定制

ASP.NET MVC 采用基于数据注解特性 (Data Annotation Attribute) 的声明式 Model 元数据定义方法, 我们将相应的数据注解特性应用在数据类型及其属性上对 Model 元数据进行定制。这些用于声明式元数据定义的特性大都定义在 System.ComponentModel.DataAnnotations.dll 程序集中, 程序集的名称同时也是对应的命名空间名称, 接下来介绍一些常用的数据注解特性以及它们对元数据的影响。

UIHintAttribute

HtmlHelper 和 HtmlHelper<TModel> 定义了一系列模板方法, 比如 Display/DisplayFor、Editor/EditorFor、DisplayForModel/EditForModel、Lable/LabelFor 和 DisplayText/DisplayTextFor。所谓模板方法, 就是说我们在通过调用这些方法将代表 Model 的数据呈现在 View 中的时候, 并不对最终呈现的 UI 元素进行显式地控制, 而采用默认或者指定的模板来决定最终呈现在浏览器中的 HTML。

每个模板均具有相应的名称, 这些模板方法在进行 Model 呈现的时候根据对应的 Model 元数据得到对应的模板名称。模板的名称通过 ModelMetadata 的 TemplateHint 属性表示, 如下面的代码片段所示, 这是一个字符串类型的可读写属性。

```

public class ModelMetadata
{
    //其他成员
    public virtual string TemplateHint { get; set; }
}

```

ModelMetadata 的 TemplateHint 属性可以通过 UIHintAttribute 特性来定制。如下面的代码片段所示, UIHintAttribute 具有 PresentationLayer 和 UIHint 两个只读属性, 分别用于限制展现层的类型 (比如 “HTML”、“Silverlight”、“WPF”、“WinForms” 和 “MVC” 等) 和模板名称, 这两个属性均在构造函数中初始化。

```

[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property,
    AllowMultiple=true)]
public class UIHintAttribute : Attribute
{
    //其他成员
    public UIHintAttribute(string uiHint);
    public UIHintAttribute(string uiHint, string presentationLayer);

    public string PresentationLayer { get; }
    public string UIHint { get; }
}

```

通过应用在 `UIHintAttribute` 上的 `AttributeUsageAttribute` 特性定义不难看出，`AllowMultiple` 属性被设置为 `True`，意味着我们可以在相同的目标元素上应用多个 `UIHintAttribute` 特性，那么哪一个会被选择用于定制 Model 元数据呢？

如果多个 `UIHintAttribute` 应用到了相同的元素（类型或者属性），会优先选择 `PresentationLayer` 属性为“Mvc”（不区分大小写）的 `UIHintAttribute`。如果这样的 `UIHintAttribute` 不存在，则选择一个 `PresentationLayer` 属性值为空的 `UIHintAttribute`。值得一提的是，如果具有多个匹配的 `UIHintAttribute` 可供选择，系统会选择第一个，但是通过反射的方式获取的“第一个特性”并不一定是最先被声明的那个特性。

接下来我们创建一个简单的实例来演示 `UIHintAttribute` 特性对 Model 元数据的定制，在本节后续部分我们同样用该实例程序测试其它数据注解特性对 Model 元数据的影响。在一个 ASP.NET MVC 应用中先创建了如下一个 `ModelMetadataInfo` 类型，它的 `ModelMetadata` 属性表示基于某个数据类型的 Model 元数据，`PropertyAccessors` 属性是一个表达式数组，每个表达式用于根据指定的 `ModelMetadata` 对象得到相应的属性值。

```
public class ModelMetadataInfo
{
    public ModelMetadata ModelMetadata { get; private set; }
    public Expression<Func<ModelMetadata, object>>[]
        PropertyAccessors { get; private set; }

    public ModelMetadataInfo(Type modelType,
        params Expression<Func<ModelMetadata, object>>[] propertyAccessors)
    {
        this.ModelMetadata = new ModelMetadata(ModelMetadataProviders.Current,
            null, null, modelType, null);
        this.PropertyAccessors = propertyAccessors;
    }
}
```

然后定义了如下一个 `HomeController`。默认的动作方法的 `Index` 中我们会根据自定义的数据类型 `DemoModel` 创建了一个 `ModelMetadataInfo` 对象，其 `PropertyAccessors` 属性包含一个根据 `ModelMetadata` 对象得到其 `TemplateHint` 属性的表达式，最终将创建的 `ModelMetadataInfo` 对象作为 Model 显示在默认的 View 中。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ModelMetadataInfo metadataInfo = new ModelMetadataInfo(typeof(DemoModel),
            metadata => metadata.TemplateHint);
        return View(metadataInfo);
    }
}

public class DemoModel
{
}
```

```

public string Foo { get; set; }

[UIHint("Template A")]
[UIHint("Template B", "Mvc")]
public string Bar { get; set; }

[UIHint("Template A")]
[UIHint("Template B")]
public string Baz { get; set; }
}

```

如上面的代码片段所示，DemoModel 具有 Foo、Bar 和 Baz 三个属性，后两个属性上应用了两个 UIHintAttribute 特性分别将模板名称设置为“Template A”和“Template B”，应用在属性 Bar 上的一个 UIHintAttribute 特性对 PresentationLayer 属性进行了设置（“Mvc”）。

然后我们为 Action 方法 Index 定义相应的 View。从如下的代码片段中（省略掉 CSS 样式设置的代码）可以看出，这是一个基于 ModelMetadataInfo 的强类型 View。在该 View 中，用于描述属性 Model 元数据的 ModelMetadata 的指定属性值通过表格的形式呈现出来。

```

@using System.Linq.Expressions
@using System.Reflection
@model ModelMetadataInfo
<html>
    <head>
        <title>Model 元数据</title>
    </head>
    <body>
        <table>
            <tr><th>Property</th>
            @foreach (Expression<Func<ModelMetadata, object>> accessor in
                Model.PropertyAccessors)
            {
                MemberExpression memberExpression = accessor.Body
                    as MemberExpression;
                if (null == memberExpression)
                {
                    UnaryExpression convertExpression = accessor.Body
                        as UnaryExpression;
                    if (null != convertExpression)
                    {
                        memberExpression =
                            (MemberExpression) convertExpression.Operand;
                    }
                }
                PropertyInfo propertyInfo =
                    (PropertyInfo) memberExpression.Member;
                <th>@propertyInfo.Name</th>
            }
        </tr>
        @foreach (ModelMetadata metadata in Model.ModelMetadata.Properties)
        {
            <tr>
                <td>@metadata.PropertyName</td>
                @foreach (Expression<Func<ModelMetadata, object>> accessor
                    in Model.PropertyAccessors)
                {
                    <td>@(accessor.Compile() (metadata) ?? "N/A")</td>

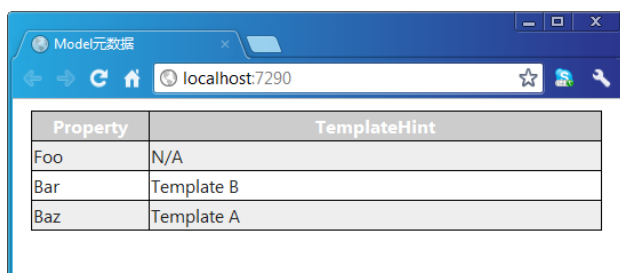
```

```

    }
  </tr>
}
</table>
</body>
</html>

```

该程序运行之后会在浏览器中呈现出如图 4-3 所示的输出结果，可以清楚地看到对于同时应用了两个 `UIHintAttribute` 特性，并对模板名称进行了不同设置的属性 `Bar` 和 `Baz`，`PresentationLayer` 被设置为“Mvc”的 `UIHintAttribute` 特性具有更高的优先级。如果多个 `UIHintAttribute` 特性具有相同的优先级，那么会选择获取的第一个 `UIHintAttribute` 特性。（S401）



Property	TemplateHint
Foo	N/A
Bar	Template B
Baz	Template A

图 4-3 针对多个 `UIHintAttribute` 特性的选择

HiddenInputAttribute 与 ScaffoldColumnAttribute

一个作为 `Model` 的数据类型有时候具有一个唯一标识(ID)，当我们以编辑模式将 `Model` 对象在 `View` 中呈现的时候，往往不允许对作为唯一标识的属性进行修改。如果 ID 不具有可读性（比如是一个随机数或者 GUID），甚至不希望让它显示在界面上，这个时候就会使用到特性 `HiddenInputAttribute`。

`HiddenInputAttribute` 并没有定义在 `System.ComponentModel.DataAnnotations` 命名空间下，它的命名空间为 `System.Web.Mvc`，所以该特性是专门为 ASP.NET MVC 设计的。顾名思义，`HiddenInputAttribute` 会将目标对象以类型为 `hidden` 的 `<input>` 元素呈现出来。在默认的情况下，应用了 `HiddenInputAttribute` 特性的目标对象依然会以只读的形式显示出来。如果不希望显示，可以将如下所示的布尔类型的 `DisplayValue` 设置为 `False`（默认值为 `True`）。

```

[AttributeUsage(AttributeTargets.Property | AttributeTargets.Class,
    AllowMultiple=false, Inherited=true)]
public sealed class HiddenInputAttribute : Attribute
{
    public HiddenInputAttribute();
    public bool DisplayValue { get;set; }
}

```

同样以前面定义的 `DemoModel` 类型为例，通过如下的方式将 `HiddenInputAttribute` 特性应用在属性 `Foo` 和 `Bar` 上，后者将 `DisplayValue` 属性设置为 `False`。

```
public class DemoModel
{
    [HiddenInput]
    public string Foo { get; set; }

    [HiddenInput(DisplayValue = false)]
    public string Bar { get; set; }

    public string Baz { get; set; }
}
```

在一个基于 DemoModel 类型的强类型 View 中, 通过如下的代码调用 HtmlHelper<TModel> 的扩展方法 EditorFor 将 DemoModel 对象三个属性 (Foo = "Foo", Bar = "Bar", Baz = "Baz") 以编辑模式呈现出来。

```
@model DemoModel
@Html.EditorFor(m=>m.Foo)
@Html.EditorFor(m=>m.Bar)
@Html.EditorFor(m=>m.Baz)
```

如下所示的是最终生成的 HTML, 可以看出应用了 HiddenInputAttribute 特性的两个属性 Foo 和 Bar 均以类型为 “hidden” 的 <input> 元素进行呈现, 不过 Foo 属性值会以文本的形式显示出来, 但 Bar 的属性值则不会。(S402)

```
Foo<input id="Foo" name="Foo" type="hidden" value="Foo" />
<input id="Bar" name="Bar" type="hidden" value="Bar" /><input class="text-box
single-line" id="Baz" name="Baz" type="text" value="Baz"/>
```

HiddenInputAttribute 针对 Model 元数据的定制体现 ModelMetadata 的如下两个属性上, 其中一个就是上面介绍的 TemplateHint, 另一个则是布尔类型的属性 HideSurroundingHtml, 它表示目标元素是否需要通过相应的 HTML 呈现在 UI 界面上。具体来说, HiddenInputAttribute 特性 ModelMetadata 对象的 TemplateHint 属性设置为 “HiddenInput”, 其 HideSurroundingHtml 属性则对应着 HiddenInputAttribute 的 DisplayValue 属性。

```
public class ModelMetadata
{
    //其他成员
    public virtual string      TemplateHint{get;set;}
    public virtual bool        HideSurroundingHtml { get; set; }
}
```

针对上面定应在 DemoModel 中的三个属性, 我们通过现有的测试程序来检测一下对应 ModelMetadata 的 TemplateHint 和 HideSurroundingHtml 属性因应用了 HiddenInputAttribute 特性而有何不同, 只需要将定义在 HomeController 的 Index 方法作如下的修改即可。

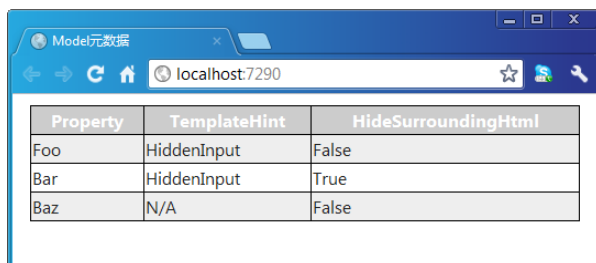
```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ModelMetadataInfo metadataInfo = new ModelMetadataInfo(typeof(DemoModel),
            metadata => metadata.TemplateHint,
```

```

        metadata => metadata.HideSurroundingHtml);
    return View(metadataInfo);
}
}

```

运行该程序会在浏览器中呈现出如图 4-4 所示的输出结果，可以看到对于应用了 `HiddenInputAttribute` 的两个属性 `Foo` 和 `Bar`，对应 `ModelMetadata` 的 `TemplateHint` 属性均为“`HiddenInput`”，而将 `DisplayValue` 属性设置为 `False` 的属性 `Bar`，其 `HideSurroundingHtml` 属性变成了 `True`。（S403）



Property	TemplateHint	HideSurroundingHtml
Foo	HiddenInput	False
Bar	HiddenInput	True
Baz	N/A	False

图 4-4 `HiddenInputAttribute` 对 Model 元数据的控制

有的读者可能会问这样一个问题，`UIHintAttribute` 和 `HiddenInputAttribute` 都会设置表示 Model 元数据的 `ModelMetadata` 对象的 `TemplateHint` 属性，如果两个特性均应用到相同的目标元素上，最终生成的 `ModelMetadata` 对象具有怎样的 `TemplateHint` 属性值呢？答案是：`UIHintAttribute` 具有更高的优先级。

对于应用了 `HiddenInputAttribute` 特性的目标元素，不论其 `DisplayValue` 具有怎样的值，都会出现在通过模板方法生成的 HTML 中。如果我们希望将它从 HTML 中移除，可以应用另一个叫作 `ScaffoldColumnAttribute` 的特性。将通过预定义模板自动生成 HTML 的方式称为“基架（Scaffolding）”，`ScaffoldColumnAttribute` 中的 `ScaffoldColumn` 代表存在于“基架”中并最终呈现在 HTML 中的字典，而该特性本身则用于控制目标元素是否应该存在于基架之中。

如下面的代码片段所示，`ScaffoldColumnAttribute` 具有一个布尔类型的只读属性 `Scaffold` 表示目标元素是否应该存在于呈现在最终生成的 HTML 的基架中，该属性在构造函数中初始化。

```

[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property,
    AllowMultiple=false)]
public class ScaffoldColumnAttribute : Attribute
{
    public ScaffoldColumnAttribute(bool scaffold);
    public bool Scaffold { get; }
}

```

`ScaffoldColumnAttribute` 最终用于控制 `ModelMetadata` 的 `ShowForDisplay` 和 `ShowForEdit` 属性。如下面的代码所示，这是两个布尔类型的属性，分别表示目标元素是否应该出现在显

示和编辑模式下的“基架”中。如果 `ShowForDisplay` 的属性为 `False`，在调用模板方法 `EditorFor/EditorForModel` 方法时目标元素将不会出现在最终生成的 HTML 中。通过 `DisplayFor/DisplayForModel` 方法生成的 HTML 将不会包含 `ShowForDisplay` 属性为 `False` 的元素。这两个属性值在默认情况下均为 `True`。

```
public class ModelMetadata
{
    //其他成员
    public virtual bool ShowForDisplay { get; set; }
    public virtual bool ShowForEdit { get; set; }
}
```

DataModelAttribute 与 DisplayFormatAttribute

用于指定数据类型的 `DataModelAttribute` 特性是经常使用的数据标注特性。这里所说的数据类型不是我们所理解的 CLR 类型，而是通过 `System.ComponentModel.DataAnnotations.DataType` 枚举表示的具有某种显示格式的数据类型。如下面的代码片段所示，`DataType` 枚举定义了一系列包括时间、日期、电话号码、货币、`Html`、电子邮箱地址在内的数据类型。

```
public enum DataType
{
    Custom,
    DateTime,
    Date,
    Time,
    Duration,
    PhoneNumber,
    Currency,
    Text,
    Html,
    MultilineText,
    EmailAddress,
    Password,
    Url,
    ImageUrl,
    CreditCard,
    PostalCode,
    Upload
}
```

为 Model 元数据设置数据类型的 `DataModelAttribute` 实际上是一个验证特性。如下面的代码片段所示，`DataModelAttribute` 直接继承自 `ValidationAttribute`。关于验证和验证特性，我们会在第 6 章“Model 的验证”中进行单独介绍。除了具有一个 `DataType` 枚举类型的 `DataType` 只读属性之外，`DataModelAttribute` 还具有一个字符串类型的表示自定义数据类型的 `CustomDataType` 属性，它们均在相应的构造函数中初始化，方法 `GetDataTypeName` 返回一个代表数据类型名称的字符串。

```
[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Field |
    AttributeTargets.Property | AttributeTargets.Method, AllowMultiple=false)]
public class DataModelAttribute : ValidationAttribute
```

```

{
    public DataTypeAttribute(DataType dataType);
    public DataTypeAttribute(string customDataType);

    public virtual string          GetDataTypeName();
    public override bool          IsValid(object value);
    public string                  CustomDataType { get; }
    public DataType                DataType { get; }
    public DisplayFormatAttribute  DisplayFormat { get; }
}

```

`DataTypeAttribute` 的只读属性 `DisplayFormat` 涉及另一个用于进行格式化的 `DisplayFormatAttribute` 特性，它可以指定一个格式化字符串以控制数据在 UI 界面上的显示格式。如下面的代码片段所示，格式化字符串通过属性 `DataFormatString` 表示，布尔类型的属性 `ApplyFormatInEditMode` 表示格式化规则是否需要应用到编辑模式，而 `HtmlEncode` 属性表示是否需要为目标内容实施 HTML 编码，默认情况下这两个属性值分别为 `False` 和 `True`。`DisplayFormatAttribute` 的属性 `NullDisplayText` 和 `ConvertEmptyStringToNull` 与空值/空字符串的处理有关，前者表示针对空值（`Null`）对象的显示文本，后者表示是否将传入的空字符串转换成 `Null`。

```

[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property,
    AllowMultiple=false)]
public class DisplayFormatAttribute : Attribute
{
    public DisplayFormatAttribute();

    public string          DataFormatString { get; set; }
    public bool            ApplyFormatInEditMode { get; set; }
    public bool            HtmlEncode { get; set; }
    public string          NullDisplayText { get; set; }
    public bool            ConvertEmptyStringToNull { get; set; }
}

```

定义在 `DataType` 枚举中的部分数据类型（比如 `Date`、`Time` 和 `Currency` 等）都具有各自的格式（它们的格式化字符串分别是 “`{0:d}`”、“`{0:t}`” 和 “`{0:C}`”）。当 `DataTypeAttribute` 通过指定的 `DataType` 枚举值被创建的时候，会根据对应的格式创建一个 `DisplayFormatAttribute` 对象作为其 `DisplayFormat` 属性值。顺便提一下，针对数据类型 `Data`、`Time` 和 `Currency` 的 `DataTypeAttribute` 对应的 `DisplayFormatAttribute`，前两个的 `ApplyFormatInEditMode` 属性为 `True`，最后一个的 `ApplyFormatInEditMode` 属性为 `False`。这也很好理解，针对日期和时间的编辑具有对应的格式。对于货币来说，编辑的时候一般就是一个无格式的数字。

`DataTypeAttribute` 和 `DisplayFormatAttribute` 对 Model 元数据的定制涉及 `ModelMetadata` 的如下属性。其中 `DataTypeAttribute` 中设置的数据类型对应于 `ModelMetadata` 的 `DataTypeName` 属性，而 `DisplayFormatAttribute` 的 `ConvertEmptyStringToNull` 和 `NullDisplayText` 属性对应着 `ModelMetadata` 的同名属性。

```
public class ModelMetadata
{
    //其他成员
    public virtual string    DataTypeName { get; set; }
    public virtual bool      ConvertEmptyStringToNull { get; set; }
    public virtual string    NullDisplayText { get; set; }
    public virtual string    DisplayFormatString { get; set; }
    public virtual string    EditFormatString { get; set; }
}
```

通过 `DisplayFormatAttribute` 的 `DataFormatString` 属性设置的格式化字符串会赋值给 `ModelMetadata` 的 `DisplayFormatString` 属性, 该属性表示显示模式下的格式化字符串。如果 `ApplyFormatInEditMode` 属性为 `True`, 该属性会赋值给 `ModelMetadata` 的 `EditFormatString` 属性, 表示编辑模式下的格式化字符串。

`ModelMetadata` 表示数据类型名称的 `DataTypeName` 属性类型为字符串, 如果 `DataTypeAttribute` 特性的 `DataType` 属性为 `Custom`, 那么以字符串设置的自定义数据类型将会作为 `ModelMetadata` 的 `DataTypeName` 属性, 否则直接将设置的 `DataType` 枚举对象转换为字符串 (直接调用 `ToString` 方法) 并直接作为 `DataTypeName` 属性值。

`DataTypeAttribute` 并不是一个封闭的 (Sealed) 类型, 可以通过继承它创建自定义的 `DataTypeAttribute`。这种情况下 `ModelMetadata` 的 `DataType` 属性是通过调用 `DataTypeAttribute` 的虚方法 `GetDataTypeName` 获取的。如果我们对 Model 元数据的数据类型名称具有特殊的定制方式, 则需要重写这个方法。

如果通过 `DataTypeAttribute` 特性以字符串的方式指定一个自定义数据类型, 该字符串将直接作为 `ModelMetadata` 的 `DataTypeName` 属性值。如果没有显式地对数据类型进行设置, 并且 `DisplayFormatAttribute` 的 `HtmlEncode` 属性为 `False` (不需要对目标内容进行 HTML 编码), 生成的 `ModelMetadata` 对象的 `DataTypeName` 属性值则为 `Html` (相当于 `DataType.Html`)。

由于有的 `DataTypeAttribute` 对应着一个 `DisplayFormatAttribute`, 如果它们同时应用在了相同的目标元素上, 在它们的设置互相冲突的情况下后者 (`DisplayFormatAttribute`) 将具有更高的优先级。

照例利用前面创建的测试程序来检测一下 `DataTypeAttribute` 和 `DisplayFormatAttribute` 对目标元素的 Model 元数据的影响。为此将 `DemoModel` 进行了如下的改写: 属性 `Foo` 和 `Bar` 分别应用了 `DataTypeAttribute` 特性将数据类型设置为预定义的 `EmailAddress` (通过 `DataType` 枚举) 和自定义的 “Barcode” (通过字符串), 属性 `Qux` 上则应用了 `DisplayFormatAttribute` 特性将 `HtmlEncode` 属性设置为 `False`。

```
public class DemoModel
{
    [DataType(DataType.EmailAddress)]
    public string Foo { get; set; }

    [DataType("Barcode")]
    public string Bar { get; set; }
}
```

```

public string Baz { get; set; }

[DisplayFormat(HtmlEncode = false)]
public string Qux { get; set; }
}

```

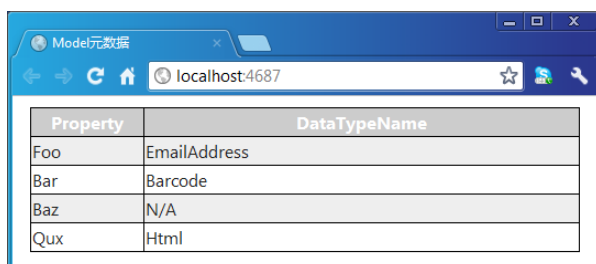
只需要对定义在 `HomeController` 的 `Index` 方法通过如下的代码让用于描述所有属性的 `ModelMetadata` 的 `DataTypeName` 属性呈现出来。

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        ModelMetadataInfo metadataInfo = new ModelMetadataInfo(typeof(DemoModel),
            metadata => metadata.DataTypeName);
        return View(metadataInfo);
    }
}

```

运行该程序后会在浏览器中呈现出如图 4-5 所示的输出结果，上面介绍的 `Model` 元数据的数据类型名称的设置规则在这里得到了很好地体现。（S404）



Property	DataTypeName
Foo	EmailAddress
Bar	Barcode
Baz	N/A
Qux	Html

图 4-5 `DataTypeAttribute/DisplayFormatAttribute` 对 `Model` 元数据数据的定制

EditableAttribute 与 ReadOnlyAttribute

`EditableAttribute` 和 `ReadOnlyAttribute` 用于控制目标元素的可读写性。如下面的代码片段所示，`EditableAttribute` 和 `ReadOnlyAttribute` 分别具有一个布尔类型的属性 `AllowEdit` 和 `IsReadOnly` 分别表示是否允许编辑和是否只读。

```

[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property,
    AllowMultiple=false, Inherited=true)]
public sealed class EditableAttribute : Attribute
{
    //其他成员
    public EditableAttribute(bool allowEdit);
    public bool AllowEdit { get; private set; }
}

[AttributeUsage(AttributeTargets.All)]
public sealed class ReadOnlyAttribute : Attribute
{

```

```
//其他成员
public ReadOnlyAttribute(bool isReadOnly);
public bool IsReadOnly { get; }
}
```

不允许编辑即为只读，所以这两个标注特性具有相同的作用。它们共同控制着 ModelMetadata 的 IsReadOnly 属性。如果同时将 EditableAttribute 和 ReadOnlyAttribute 应用到相同的目标元素上并且作出相反的设置（让 EditableAttribute 的 AllowEdit 属性和 ReadOnlyAttribute 的 IsReadOnly 属性具有相同的布尔值），EditableAttribute 特性具有更高的优先级。

```
public class ModelMetadata
{
    //其他成员
    public virtual bool IsReadOnly{get; set;}
}
```

通过如下方式将特性 EditableAttribute 和 ReadOnlyAttribute 同时应用到类型 DemoModel 的 Bar 和 Baz 属性上，并在读写性上作出相反的设置，而在属性 Foo 上应用了 ReadOnlyAttribute 特性并将其设为只读。

```
public class DemoModel
{
    [ReadOnly(true)]
    public string Foo { get; set; }

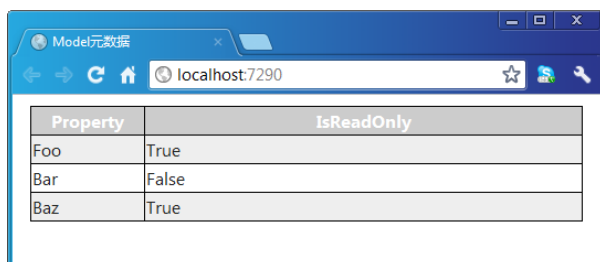
    [Editable(true)]
    [ReadOnly(true)]
    public string Bar { get; set; }

    [Editable(false)]
    [ReadOnly(false)]
    public string Baz { get; set; }
}
```

然后对定义在 HomeController 中的 Index 方法作了如下的改动，使 DemoModel 的三个属性对应 ModelMetadata 的 IsReadOnly 属性显示出来。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ModelMetadataInfo metadataInfo = new ModelMetadataInfo(typeof(DemoModel),
            metadata => metadata.IsReadOnly);
        return View(metadataInfo);
    }
}
```

上面的程序运行之后会在浏览器上呈现出如图 4-6 所示的输出结果。由于 Foo 属性上仅仅应用了 ReadOnlyAttribute 特性，所以它控制了 ModelMetadata 的 IsReadOnly 属性。而 Bar 和 Baz 属性则同时应用 EditableAttribute 和 ReadOnlyAttribute 两个特性，ModelMetadata 的 IsReadOnly 属性最终通过 EditableAttribute 特性来控制。（S405）



Property	IsReadOnly
Foo	True
Bar	False
Baz	True

图 4-6 EditableAttribute/ReadOnlyAttribute 对 Model 元数据的定制

DisplayAttribute 与 DisplayNameAttribute

DisplayAttribute 特性为目标元素定义一些说明性文字。如下面的代码片段所示，**DisplayAttribute** 具有 5 个基本属性，其中 **Name** 和 **ShortName** 是为目标元素设置一个显示名称和一个简短的显示名称。属性 **Description** 和 **Order** 为目标元素设置描述性文字和用于排序的权重。字符串类型的 **Prompt** 属性为目标元素设置一个字符串，它在 UI 界面上以水印的方式呈现。

```
[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Field |
    AttributeTargets.Property | AttributeTargets.Method, AllowMultiple=false)]
public sealed class DisplayAttribute : Attribute
{
    //其他成员
    public DisplayAttribute();

    public string GetName();
    public string GetShortName();
    public string GetDescription();
    public int? GetOrder();
    public string GetPrompt();

    public string Name { get; set; }
    public string ShortName { get; set; }
    public string Description { get; set; }
    public int Order { get; set; }
    public string Prompt { get; set; }

    public Type ResourceType { get; set; }
}
```

由于 **DisplayAttribute** 特性设置的文字都是面向最终用户的，所以有必要对其进行本地化（Localization），为此该特性允许我们通过资源文件的方式来定义它们。**DisplayAttribute** 特性的 **ResourceType** 代表采用的资源文件生成的类型。如果我们对该属性进行了显式设置，上述 5 个属性值将会被认为是对应的资源条目的名称。正因为如此，如果我们需要得到最终用于显示的文字，不能通过相应的属性而需要通过相应的 **GetXxx** 方法来获取。

另一个定义在命名空间 **System.ComponentModel** 下的 **DisplayNameAttribute** 特性则专门用于设置目标元素的显示名称。如下面的代码片段所示，目标元素的显示名称通过只读属性

DisplayName 表示, 该属性在构造函数中被初始化。如果调用默认的构造函数, 该属性会被设置为空字符串。

```
[AttributeUsage(AttributeTargets.Event | AttributeTargets.Property |
    AttributeTargets.Method | AttributeTargets.Class)]
public class DisplayNameAttribute : Attribute
{
    public DisplayNameAttribute();
    public DisplayNameAttribute(string displayName);

    public virtual string DisplayName { get; }
}
```

DisplayAttribute 和 DisplayNameAttribute 特性对 Model 元数据的定制涉及五个属性。DisplayAttribute 的 GetName 方法的返回值和 DisplayNameAttribute 的属性 DisplayName 对应于 ModelMetadata 的 DisplayName 属性。DisplayAttribute 的 GetShortName 方法则获取 ModelMetadata 的 ShortDisplayName 属性。而 GetDescription 和 GetOrder 方法返回 ModelMetadata 的 Description 和 Order 属性 (默认值为 10000)。ModelMetadata 的 Watermark 属性通过 DisplayAttribute 的 GetPrompt 方法的返回值来初始化。

```
public class ModelMetadata
{
    //其他成员
    public virtual string DisplayName { get; set; }
    public virtual string ShortDisplayName { get; set; }
    public virtual string Description { get; set; }
    public virtual int Order { get; set; }
    public virtual string Watermark { get; set; }
}
```

由于 DisplayAttribute 的 GetName 方法的返回值和 DisplayNameAttribute 的 DisplayName 属性最终都用于设置 ModelMetadata 的 DisplayName 属性, 如果这两个属性同时应用到相同的目标元素上并且对显示名称作出了不同的设置, 那么 DisplayAttribute 特性具有更高的优先级。

如下面的代码片段所示, 我们将 DisplayAttribute 和 DisplayNameAttribute 特性应用到了定义在测试程序中的数据类型 DemoModel 的相应的属性上。其中属性 Bar 上应用了 DisplayNameAttribute 并将显示名称设置为“Bar”, 而属性 Baz 上同时应用了 DisplayAttribute 和 DisplayNameAttribute 特性并分别将显示名称设置为“BAZ”和“baz”。应用在属性 Bar 上的 DisplayAttribute 还对其他相关属性作了相应的设置。

```
public class DemoModel
{
    public string Foo { get; set; }

    [DisplayName("Bar")]
    public string Bar { get; set; }

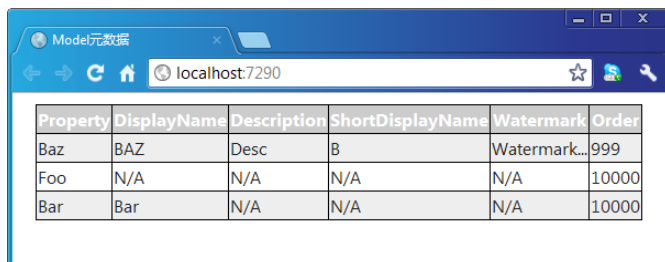
    [Display(Name = "BAZ", Description = "Desc",
        ShortName="B", Prompt="Watermark...", Order=999)]
```

```
[DisplayName("baz")]
public string Baz { get; set; }
}
```

我们对定义在 HomeController 的 Index 方法作了如下的改动，使基于属性的 ModelMetadata 的与 DisplayAttribute/DisplayNameAttribute 特性相关的属性显示出来。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ModelMetadataInfo metadataInfo = new ModelMetadataInfo(typeof(DemoModel),
            metadata => metadata.DisplayName,
            metadata => metadata.Description,
            metadata => metadata.ShortDisplayName,
            metadata => metadata.Watermark,
            metadata => metadata.Order);
        return View(metadataInfo);
    }
}
```

上面的程序运行之后会在浏览器上呈现出如图 4-7 所示的输出结果。可以看到对于同时应用了 DisplayAttribute 和 DisplayNameAttribute 特性的 Baz 属性，对应 ModelMetadata 的 DisplayName 属性与 DisplayAttribute 是一致的。对于没有通过 DisplayAttribute 特性对 Order 进行设置的属性 Foo 和 Baz，该属性默认为 10000。（S406）



Property	DisplayName	Description	ShortDisplayName	Watermark	Order
Baz	BAZ	Desc	B	Watermark...	999
Foo	N/A	N/A	N/A	N/A	10000
Bar	Bar	N/A	N/A	N/A	10000

图 4-7 DisplayAttribute/DisplayNameAttribute 对 Model 元数据的定制

RequiredAttribute

我们来介绍最终一个标注特性 RequiredAttribute。顾名思义，RequiredAttribute 特性将目标元素设置为必需的数据成员。如下面的代码片段所示，RequiredAttribute 和 DataTypeAttribute 是一个验证特性。其 AllowEmptyStrings 属性表示作为必需数据成员的目标元素是否接受一个空字符串，默认情况下是不允许的。

```
[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Field |
    AttributeTargets.Property, AllowMultiple=false)]
public class RequiredAttribute : ValidationAttribute
{
    public RequiredAttribute();
    public bool AllowEmptyStrings { get; set; }
}
```


对于应用了 `RequiredAttribute` 特性的数据成员，对应 `ModelMetadata` 的 `IsRequired` 属性将会被设置为 `True`。如下面的代码片段所示，该属性是一个可读写的属性。

```
public class ModelMetadata
{
    //其他成员
    public virtual bool IsRequired { get; set; }
}
```

4.1.4 IMetadataAware 接口

在介绍用于设置 Model 元数据自定义属性的 `AdditionalMetadataAttribute` 特性时，我们提到了它实现的接口 `IMetadataAware`，这是一个非常重要并且有用的接口，通过自定义实现该接口的特性可以对最终生成的 Model 元数据进行自由地定制。如下面的代码片段所示，`IMetadataAware` 接口具有唯一的方法成员 `OnMetadataCreated`，针对作为参数的 `ModelMetadata` 对象的定制就体现在该方法中。

```
public interface IMetadataAware
{
    void OnMetadataCreated(ModelMetadata metadata);
}
```

当 Model 元数据被创建出来后，上述的这一系列数据注解特性会被提取出来对其进行初始化，然后获取应用在目标元素上所有实现了 `IMetadataAware` 接口的特性，并将初始化的 `ModelMetadata` 对象作为参数调用 `OnMetadataCreated` 方法。通过自定义实现该接口的特性不仅仅可以添加一些额外的元数据属性，还可以修改已经通过相应的标注特性初始化的相关属性。

ASP.NET MVC 定义了两个实现了 `IMetadataAware` 接口的特性，一个就是已经介绍过的 `AdditionalMetadataAttribute`，另一个则是 `System.Web.Mvc.AllowHtmlAttribute`。

AllowHtmlAttribute

出于安全考虑，ASP.NET MVC 在进行 Model 绑定过程中会对请求数据进行验证以确保没有任何 HTML 标记被包含其中。`ModelMetadata` 的 `RequestValidationEnabled` 属性开启/关闭请求验证的开关。该属性在默认情况下为 `True`，意味着默认开启针对 HTML 标记的请求验证。

```
public class ModelMetadata
{
    //其他成员
    public virtual bool RequestValidationEnabled { get; set; }
}
```

如果在数据类型或者属性上应用了 `AllowHtmlAttribute` 特性，意味着允许绑定到目标元

素的原始内容包含 HTML 标记, 换言之需要忽略针对请求的验证。如下面的代码片段所示, `AllowHtmlAttribute` 是实现了 `IMetadataAware` 接口, 在 `OnMetadataCreated` 方法中它直接将作为参数的 `ModelMetadata` 对象的 `RequestValidationEnabled` 属性设置为 `False`, 从而使针对目标对象的请求验证被忽略掉。

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple=false, Inherited=true)]
public sealed class AllowHtmlAttribute : Attribute, IMetadataAware
{
    public void OnMetadataCreated(ModelMetadata metadata)
    {
        //其他操作
        metadata.RequestValidationEnabled = false;
    }
}
```

为了验证 ASP.NET MVC 针对 HTML 标记的请求验证和 `AllowHtmlAttribute` 的作用, 我们来做一个简单的实例演示。在一个 ASP.NET MVC 应用中定义了如下一个数据类型 `DemoModel`, `DemoModel` 定义了两个字符串类型的属性 `Foo` 和 `Bar`, 后者应用了 `AllowHtmlAttribute` 特性。

```
public class DemoModel
{
    public string Foo { get; set; }

    [AllowHtml]
    public string Bar { get; set; }
}
```

然后创建如下一个默认的 `HomeController`, 默认的 `Action` 方法 `Index` 中具有一个类型为 `DemoModel` 的参数, 该参数直接作为 `Model` 呈现在默认的 `View` 中。

```
public class HomeController : Controller
{
    public ActionResult Index(DemoModel model)
    {
        return View(model);
    }
}
```

如下所示的是 `Action` 方法 `Index` 所对应 `View` 的定义, 这是一个以 `Foo` 为 `Model` 的强类型 `View`。在该 `View` 中, 直接调用 `HtmlHelper<Model>` 的 `EditorForModel` 方法将 `Foo` 对象以编辑模式呈现出来。

```
@model DemoModel
<html>
    <head>
        <title>AllowHtml</title>
    </head>
    <body>
        @Html.EditorForModel()
    </body>
</html>
```

现在直接运行该 Web 应用。根据 Model 绑定的规则我们知道，如果通过浏览器访问 HomeController 的 Index 操作，可以分别指定名称为 Foo 和 Bar 的查询字符串对作为参数的 DemoModel 对象的两个属性进行初始化。为了验证对包含 HTML 标记的输入的验证，将最终绑定到 Model 上的查询字符串设置为“<script></script>”。

如图 4-8 所示，由于属性 Bar 上应用了 AllowHtmlAttribute 特性使之支持包含 HTML 标记的数据，所以以查询字符串方式指定的包含 HTML 标记的内容（“<script></script>”）直接显示在相应的文本框中。但是 Bar 属性在默认情况下不允许绑定的数据具有任何 HTML 标记，所以会将输入的数据视为恶意注入的 HTML，直接抛出异常。（S407）

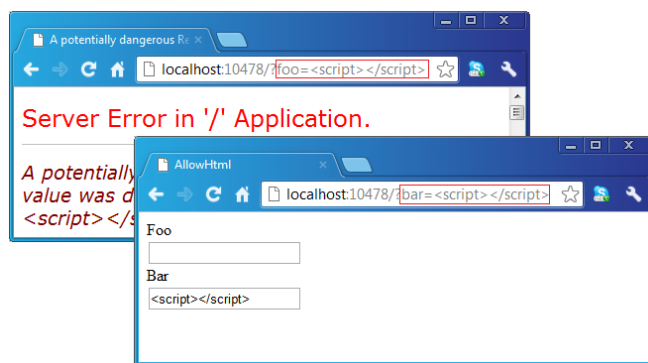


图 4-8 针对 HTML 标记请求的验证与 AllowHtmlAttribute 作用

实例演示：创建实现 IMetadataAware 接口的特性定制 Model 元数据（S408）

我们知道，数据项显示名称可以通过在数据类型或者属性成员上应用 DisplayAttribute 特性来定义，在使用该特性的时候，需要显式指定表示显示名称的 Name 属性。如果需要进行本地化处理，需要将显示内容定义在某个资源文件中，并通过 ResourceType 属性指定该资源文件生成的类型。

为了简化，通过实现 IMetadataAware 接口的方式定义了如下一个 DisplayTextAttribute 特性。该特性的属性 DisplayName/ResourceType 与 DisplayAttribute 的 Name/ResourceType 具有相同的作用，唯一不同的是 DisplayTextAttribute 的这两个属性均是可以缺省的。如果 DisplayName 没有显式指定，则默认使用属性名称或者类型名称。如果 ResourceType 没有显式指定，则采用通过静态字段 staticResourceType 表示的默认资源类型，该类型通过静态方法 SetResourceType 进行注册。

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Property)]
public class DisplayTextAttribute : Attribute, IMetadataAware
{
    private static Type staticResourceType;
    public string DisplayName { get; set; }
    public Type ResourceType { get; set; }
}
```

```

public DisplayTextAttribute()
{
    this.ResourceType = staticResourceType;
}

public void OnMetadataCreated(ModelMetadata metadata)
{
    this.DisplayName = this.DisplayName ??
        (metadata.PropertyName ?? metadata.ModelType.Name);
    if (null == this.ResourceType)
    {
        metadata.DisplayName = this.DisplayName;
        return;
    }
    PropertyInfo property = this.ResourceType.GetProperty(this.DisplayName,
        BindingFlags.NonPublic|BindingFlags.Public| BindingFlags.Static);
    metadata.DisplayName = property.GetValue(null, null).ToString();
}

public static void SetResourceType(Type resourceType)
{
    staticResourceType = resourceType;
}
}

```

`DisplayTextAttribute` 对 Model 元数据的定制实现在 `OnMetadataCreated` 方法中。该方法根据设置的 `DisplayName` 和 `ResourceType` 属性解析出最终作为显示名称的文本，并作为 `ModelMetadata` 的 `DisplayName` 属性值。

接下来演示如何使用这个 `DisplayTextAttribute` 特性来替换 `DisplayAttribute` 特性进行显示名称的设置。我们在一个 ASP.NET MVC 应用中定义如下一个表示员工的 `Employee` 类型。`Employee` 所有的属性上均应用了 `DisplayTextAttribute` 特性，而 `DisplayName` 和 `RerourceType` 属性没有显式指定。

```

public class Employee
{
    [DisplayText]
    public string Name { get; set; }

    [DisplayText]
    public string Gender { get; set; }

    [DisplayText]
    [DataType(DataType.Date)]
    public DateTime BirthDate { get; set; }

    [DisplayText]
    public string Department { get; set; }
}

```

接下来打开项目的属性对话框并选择“资源 (Resources)” Tab 页，按照如图 4-9 所示为 `Employee` 中的四个属性定义相应的资源字符串作为显示的名称，资源字符串条目的名称为属性名。

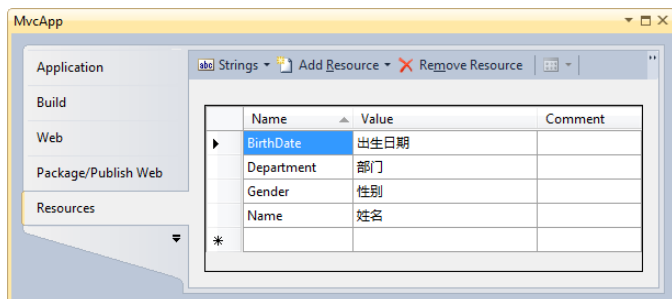
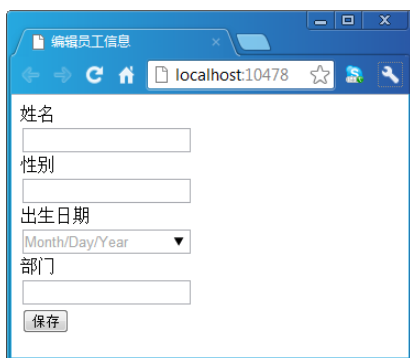


图 4-9 基于属性名称的资源字符串的定义

该资源文件会自动生成一个类型为 `Resources` 的内部类型。由于应用在 `Employee` 属性上的 `DisplayTextAttribute` 特性并没有显式指定资源类型，所以需要在 `Global.asax` 文件中通过如下方式将 `Resources` 类型注册为默认的资源类型。

```
public class MvcApplication : System.Web.HttpApplication
{
    //其他成员
    protected void Application_Start()
    {
        //其他操作
        DisplayTextAttribute.SetResourceType(typeof(Resources));
    }
}
```

现在通过调用 `HtmlHelper<TModel>` 的 `EditorForModel` 方法将一个具体的 `Employee` 对象以编辑模式显示在某个 `Model` 类型为 `Employee` 的强类型 `View` 上，会呈现出如图 4-10 所示的效果。可以看到，作为标签显示的文字正是我们定义在资源文件中的内容。

图 4-10 `Employee` 对象在编辑模式下默认的呈现效果

4.2 Model 元数据与 Model 模板

`Model` 元数据的一个主要的作用就是为定义在 `HtmlHelper<TModel>` 中的模板方法（这

些模板方法包括 `Display/DisplayFor`、`Editor/EditorFor`、`DisplayForModel/EditForModel`、`Label/LabelFor` 和 `DisplayText/DisplayTextFor` 等) 提供辅助生成 HTML 的元数据信息。

在调用这些方法的时候, 如果指定了一个具体的通过 `Partial View` 定义的模板, 或者对应的 `ModelMetadata` 的 `TemplateHint` 属性被设置了相应的模板名称, 会自动采用该模板来生成最终的 HTML。如果没有指定模板名称, 则会根据数据类型在预定义的目录下去寻找作为模板的 `Partial View`。为了让读者对模板及其作用有一个大体的认识, 我们来做一个简单的实例演示。

4.2.1 实例演示: 通过模板将布尔值显示为 RadioButton (S409)

在默认的情况下, 不论是对于编辑模式还是显示模式, 一个布尔类型的属性值总是以一个 `CheckBox` 的形式呈现出来。创建如下一个表示员工的类型 `Employee`, 他具有一个布尔类型的属性 `IsPartTime` 表示该员工是否为兼职。

```
public class Employee
{
    [DisplayName("姓名")]
    public string Name { get; set; }

    [DisplayName("部门")]
    public string Department { get; set; }

    [DisplayName("是否兼职")]
    public bool IsPartTime { get; set; }
}
```

如果直接调用 `HtmlHelper<TModel>` 的 `EditorForModel` 方法将一个 `Employee` 对象显示在某个 `Model` 类型为 `Employee` 的强类型 `View` 中, 最终会呈现出如图 4-11 的所示的效果, 可以看到, 表示是否为兼职的 `IsPartTime` 属性最终以 `CheckBox` 形式被呈现出来。

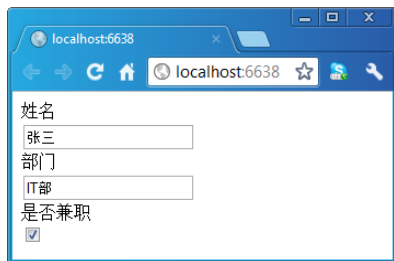


图 4-11 默认模板对布尔值的呈现效果

现在我们希望的是将所有布尔类型对象显示为两个 `RadioButton`, 具体的显示效果如图 4-12 所示。可以通过创建一个 `Model` 类型为 `Boolean` 的 `Partial View` 来创建一个模板, 使之改变所有布尔类型对象的默认呈现效果。



图 4-12 自定义模板对布尔值的呈现效果

由于我们需要改变的是布尔类型对象在编辑模式下的呈现形式，所以需要将作为模板的 View 定义在 EditorTemplates 目录下。这个目录可以存在于“Views/Shared”下，也可以存在于“Views/{ControllerName}”下。由于 ASP.NET MVC 是采用数据类型作为匹配条件来寻找对应的模板的，所以我们需要将模板 View 命名为 Boolean。下面的代码片段体现了整个 Partial View 的定义，通过调用 HtmlHelper 的 RadioButton 方法将两个布尔值（True/False）映射为对应的 RadioButton，并且采用<table>来布局。

```
@model bool
<table>
  <tr>
    <td>@Html.RadioButton("",true,Model) 是</td>
    <td>@Html.RadioButton("",false,!Model) 否</td>
  </tr>
</table>
```

值得一提的是，我们没有指定 RadioButton 的名称，而是指定一个空字符串。Html 本身会对其进行命名，而命名的依据就是 Model 元数据。Employee 的 IsPartTime 属性呈现在界面上对应的 HTML 如下所示，可以看到两个类型为 radio 的<input>元素的 name 被自动赋上了对应的属性名称。美中不足的是它们具有相同的 ID，如果希望让 ID 具有唯一性，可以对模板进行更加细致的定制。

```
<div class="editor-label"><label for="IsPartTime">是否兼职</label></div>
<div class="editor-field">
  <table>
    <tr>
      <td><input checked="checked" id="IsPartTime" name="IsPartTime"
        type="radio" value="True" .../>是</td>
      <td><input id="IsPartTime" name="IsPartTime" type="radio"
        value="False" />否</td>
    </tr>
  </table>
</div>
```

4.2.2 预定义模板

上面我们介绍了如何通过 Partial View 的方式创建模板，进而控制某种数据类型或者某

个目标元素对应的呈方式。实际上在 ASP.NET MVC 的内部定义了一系列的预定义模板。当我们调用 `HtmlHelper/HtmlHelper<TModel>` 的模板方法对 Model 或者 Model 的某个成员进行呈现的时候，系统会根据当前的呈现模式（显示模式和编辑模式）和 Model 元数据获取一个具体的模板（自定义模板者预定义模板。由于 Model 具有显示和编辑两种呈现模式，所以定义在 ASP.NET MVC 内部的默认模板可以划分为这两种基本的类型。接下来就逐个介绍这些预定义模板及最终的 HTML 呈现方式。

EmailAddress

该模板专门针对于表示 Email 地址的字符串类型的数据成员，它将目标元素呈现为一个 href 属性具有“mailto:”前缀的链接（<a>）。由于该模板仅仅用于 Email 地址的显示，所以只在显示模式下有效，或者说 ASP.NET MVC 仅仅定义了基于显示模式的 EmailAddress 模板。为了演示数据在不同模板下的呈现方式，定义了如下一个简单的数据类型 Model，通过在属性 Foo 上应用 `UIHintAttribute` 特性将模板名称设置为“EmailAddress”。

```
public class DemoModel
{
    [UIHint("EmailAddress")]
    public string Foo { get; set; }
}
```

然后在一个基于 DemoModel 类型的强类型 View 中，通过调用 `HtmlHelper<TModel>` 的 `DisplayFor` 方法将一个具体的 Model 对象的 Foo 属性以显示模式呈现出来。

```
@model Model
@Html.DisplayFor(m=>m.Foo)
```

如下的代码片段表示 Model 的 Foo 属性对应的 HTML，可以看到它就是一个针对 E-mail 地址的链接。当我们点击该链接的时候，相应的 E-mail 编辑软件（比如 Outlook）会被开启用于针对目标 E-mail 地址的邮件编辑。

```
<a href="mailto:foo@gmail.com">foo@gmail.com</a>
```

HiddenInput

关于默认模板 `HiddenInput` 我们不应该感到陌生，前面介绍的 `HiddenInputAttribute` 特性就是将 `ModelMetadata` 对象的 `TemplateHint` 属性设置为“`HiddenInput`”。如果目标元素采用 `HiddenInput` 模板，在显示模式下内容会以文本的形式显示；在编辑模式下不仅会以文本的方式显示其内容，还会生成一个对应的类型“hidden”的<input>元素。如果表示 Model 元数据的 `ModelMetadata` 对象的 `HideSurroundingHtml` 属性为 `True`（将应用在目标元素上的特性 `HiddenInputAttribute` 的 `DisplayValue` 属性设置为 `False`），不论是显示模式还是编辑模式下显示的文本都将消失。

同样以上面定义的数据类型 DemoModel 为例，通过在 Foo 属性上应用 `UIHintAttribute`

特性将模板名称设置为“HiddenInput”。

```
public class DemoModel
{
    [UIHint("HiddenInput")]
    public string Foo { get; set; }
}
```

然后在一个基于 DemoModel 类型的强类型 View 中分别调用 HtmlHelper<TModel> 的 DisplayFor 和 EditFor 方法将一个具体的 Model 对象的 Foo 属性以显示和编辑模式呈现出来。

```
@model Model
@Html.DisplayFor(m=>m.Foo)
@Html.EditorFor(m=>m.Foo)
```

分别以两种模式呈现出来的 Foo 属性对应的 HTML 如下（包含在花括号中的 GUID 表示属性值）。第一行是针对显示模式的，可以看出最终呈现出来仅限于表示属性值的文本。编辑模式对应的 HTML 中不仅包含属性值文本，还具有一个对应的类型为“hidden”的<input> 元素。

```
{42A1E9B7-2AED-4C8E-AB55-78813FC8C233}
{42A1E9B7-2AED-4C8E-AB55-78813FC8C233}<input id="Foo" name="Foo"
type="hidden" value="{42A1E9B7-2AED-4C8E-AB55-78813FC8C233}" />
```

现在我们对数据类型 Model 做一下简单修改，将应用在属性 Foo 上的 UIHintAttribute 特性替换成 HiddenInputAttribute 特性，并将其 DisplayValue 属性设置成 False。

```
public class DemoModel
{
    [HiddenInput(DisplayValue = false)]
    public string Foo { get; set; }
}
```

由于应用在目标元素上的 HiddenInputAttribute 特性的 DisplayValue 属性会最终控制对应 ModelMetadata 的 HideSurroundingHtml 属性，而后者控制是否需要生成用于显示目标内容的 HTML，所以针对的 Model 定义，最终会生成如下一段 HTML。

```
<input id="Foo" name="Foo" type="hidden" value="{42A1E9B7-2AED-4C8E-AB55-78813FC8C233}" />
```

Html

如果目标对象的内容包含一些 HTML，并需要在 UI 界面中原样呈现出来，我们可以采用 Html 模板。和 EmailAddress 模板一样，该模板仅限于显示模式。为了演示 Html 模板对目标内容的呈现方法与默认呈现方式之间的差异，我们定义了如下一个数据类型 DemoModel。该数据类型具有两个字符串类型的属性 Foo 和 Bar，其中 Foo 上面应用 UIHintAttribute 特性将模板名称设置为“Html”。

```
public class DemoModel
{
    [UIHint("Html")]
    public string Foo { get; set; }
    public string Bar { get; set; }
}
```

现在我们创建一个具体的 `DemoModel` 对象，并将 `Foo` 和 `Bar` 设置为一段表示链接的文本(google.com), 最终在一个基于 `Model` 类型的强类型 `View` 中通过调用 `HtmlHelper<TModel>` 的 `DisplayFor` 方法将这两个属性以显示模式呈现出来。

```
@model DemoModel
@Html.DisplayFor(m=>m.Foo)
@Html.DisplayFor(m => m.Bar)
```

从如下所示的表示 `Foo` 和 `Bar` 两属性的 HTML 中我们不难看出：采用 `Html` 模板的 `Foo` 属性的内容原样输出，而包含在属性 `Bar` 中的 HTML 都进行了相应的编码。

```
<a href="www.google.com">google.com</a>
&lt;a href=&quot;www.google.com&quot;&gt;google.com&lt;/a&gt;
```

Text 与 String

不论是在显示模式还是编辑模式，`Text` 和 `String` 着两个模板具有相同的 HTML 呈现方式（实际上在 ASP.NET MVC 内部，两种模板终生成的 HTML 是通过相同的方法产生的）。对于这两种模板说，目标内容在显示模式下直接以文本的形式输出，而在编辑模式下则对应着一个单行的文本框。

为了演示两种模板的同一性，我们对上面定义的数据类型 `DemoModel` 略作修改，在属性 `Foo` 和 `Bar` 上应用 `UIHintAttribute` 特性并将模板称分别设置为 `String` 和 `Text`。

```
public class DemoModel
{
    [UIHint("String")]
    public string Foo { get; set; }
    [UIHint("Text")]
    public string Bar { get; set; }
}
```

然后创建一个具体的 `DemoModel` 对象，将其作为 `Model` 呈现在具有如下定义的 `View` 中。这是一个 `Model` 类型为 `DemoModel` 的强类型 `View` 中，在该 `View` 中分别调用 `HtmlHelper<TModel>` 的 `DisplayFor` 和 `EditorFor` 方法将 `Model` 对象的两个属性以显示和编辑模式呈现出来。

```
@model DemoModel
@Html.DisplayFor(m=>m.Foo)
@Html.DisplayFor(m => m.Bar)
@Html.EditorFor(m=>m.Foo)
@Html.EditorFor(m => m.Bar)
```

如下所示的代码片段体现了上述四个元素对应的 HTML (“Dummy text...” 是 `Foo` 和 `Bar`

的属性值), 可以看到采用了 Text 和 String 模板的两个属性在显示和编辑模式下具有相同的 HTML 输出。编辑模式下输出的类型为“text”的<input>元素, 表示 CSS 特性类型的 class 属性被设置为“text-box single-line”, 意味着这是一个基于单行的文本框。

```
Dummy text ...
Dummy text ...
<input class="text-box single-line" id="Foo" name="Foo" type="text"
value="Dummy text ..." />
<input class="text-box single-line" id="Bar" name="Bar" type="text"
value="Dummy text ..." />
```

值得一提的是, ASP.NET MVC 内部采用基于类型的模板匹配机制, 对于字符串类型的数据成员, 如果没有显式设置采用的模板名称, 默认情况下会采用 String 模板。

Url

与 EmailAddress 和 Html 一样, 模板 Url 也仅限于显示模式。对于某个表示为 Url 的字符串, 如果我们希望它最终以一个链接的方式呈现在最终生成的 HTML 中, 可以选择该模板。如下面的代码片段所示, 通过应用 UIHintAttribute 特性将模板 Url 应用到属性 Foo 中。

```
public class DemoModel
{
    [UIHint("Url")]
    public string Foo { get; set; }
}
```

创建一个具体的 DemoModel 对象, 并将 Foo 属性设置为一个表示 Url 的字符串“http://www.asp.net”, 最后通过如下的方式将该属性以显示模式呈现出来。

```
@model DemoModel
@Html.DisplayFor(m=>m.Foo)
```

如下面的代码片段所示, 该属性最终呈现为一个 href 属性和文本内容均为指定属性值的链接 (<a>...)。

```
<a href="http://www.asp.net">http://www.asp.net</a>
```

MultilineText

一般的字符串在编辑模式下会呈现为一个单行的文本框(类型为“text”的<input>元素), 而 MultilineText 模板会将表示目标内容的字符串通过一个<textarea>元素来呈现, 该模板仅限于编辑模式。如下面的代码片段所示, 通过在字符串类型的 Foo 属性上应用 UIHintAttribute 特性将应用的模板设置为 MultilineText。

```
public class DemoModel
{
    [UIHint("MultilineText")]
    public string Foo { get; set; }
}
```

现在创建一个具体的 `DemoModel` 对象并通过如下的形式将 `Foo` 属性以编辑模式呈现在某个基于 `Model` 类型的强类型 `View` 中。

```
@model DemoModel
@Html.EditorFor(m=>m.Foo)
```

如下所示的代码片段表示 `DemoModel` 的 `Foo` 属性呈现在 UI 界面中的 HTML (“dummy text...” 是 `Foo` 的属性值)，可以看到这是一个 `<textarea>` 元素。表示 CSS 样式类型的 `class` 属性被设置为 “text-box multi-line”，意味着它是以多行的效果呈现。

```
<textarea class="text-box multi-line" id="Foo" name="Foo">dummy
text ...</textarea>
```

Password

对于表示密码的字符串来说，在编辑模式下应该呈现为一个类型为 “password” 的 `<input>` 元素，以使我们输入的内容以掩码的形式显示出来以保护密码的安全性。在这种情况下可以采用 `Password` 模板，该模板和 `MultilineText` 一样也仅限于编辑模式。如下面的代码片段所示，我们在 `DemoModel` 的 `Foo` 属性上应用 `UIHintAttribute` 特性将模式名称设置为 “Password”。

```
public class DemoModel
{
    [UIHint("Password")]
    public string Foo { get; set; }
}
```

创建一个具体的 `DemoModel` 对象，并通过如下的形式将 `Foo` 属性以编辑模式呈现在某个基于 `DemoModel` 的强类型 `View` 中。

```
@model DemoModel
@Html.EditorFor(m=>m.Foo)
```

该 `Foo` 属性最终会以如下的形式通过一个类型为 “Password” 的 `<input>` 元素呈现出来，表示 CSS 样式类型的 `class` 属性被设置为 “text-box single-line password”，意味着呈现效果为一个单行的文本框。

```
<input class="text-box single-line password" id="Foo" name="Foo" type=
"password" value="" />
```

Decimal

如果采用 `Decimal` 模板，代表目标元素的数字不论其小数位数是多少，都会最终被格式化为两位小数。在显示模式下，被格式化的数字直接以文本的形式呈现出来，而在编辑模式下则对应着一个单行的文本框。如下面的代码片段所示，我们在数据类型 `DemoModel` 中定义了两个对象类型属性 `Foo` 和 `Bar`，它们应用了 `UIHintAttribute` 特性并将模板名称指定为 “Decimal”。

```
public class DemoModel
{
    [UIHint("Decimal")]
    public object Foo { get; set; }

    [UIHint("Decimal")]
    public object Bar { get; set; }
}
```

我们来创建一个具体的 DemoModel 对象，将它的 Foo 和 Bar 属性分别设置为整数 123 和浮点数 3.1415（4 位小数），最终通过如下的形式将它们以显示和编辑的模式呈现在一个基于 Model 类型的强类型 View 中。

```
@model DemoModel
@Html.DisplayFor(m=>m.Foo)
@Html.DisplayFor(m=>m.Bar)
@Html.EditorFor(m=>m.Foo)
@Html.EditorFor(m=>m.Bar)
```

上述四个元素在最终呈现的 UI 界面中对应着如下的 HTML 代码，可以看到最终显示的都是具有两位小数的数字。

```
123.00
3.14
<input class="text-box single-line" id="Foo" name="Foo" type="text"
    value="123.00" />
<input class="text-box single-line" id="Bar" name="Bar" type="text"
    value="3.14"/>
```

Boolean

通过本章最开始的实例演示我们知道，一个布尔类型的对象在编辑模式下会以一个类型为“checkbox”的<input>元素的形式呈现，实际上在显示模式下它依然对应着这么一个元素，只是其 disabled 属性会被设置为 True 使之处于只读状态。布尔类型的这种默认呈现方式源自“Boolean”模板默认被使用。

当布尔类型的目标元素以编辑模式进行呈现的时候，除了生成一个类型为“checkbox”的<input>元素之外还会附加产生一个类型为“hidden”的<input>元素。如下面的代码片段所示，这个 hidden 元素具有与 CheckBox 相同的名称，但是值为 False，它存在的目的在于当 CheckBox 没有被勾选的情况下，通过对应的 hidden 元素向服务器提交相应的值（False），因为没有被勾选的 CheckBox 的值是不会包含在请求中的。

```
<input id="Foo" name="Foo" type="checkbox" value="true" />
<input name="Foo" type="hidden" value="false" />
```

Boolean 和 String、Decimal 以及后面我们将要介绍的 Object 都属于基于 CLR 类型的模板，由于 ASP.NET 在内部采用基于类型的模板匹配策略，如果没有显示设置采用的模板类型，相应类型的元素会默认采用与之匹配的模板。

Collection

顾名思义，Collection 模板用于集合类型的目标元素的显示与编辑。对于采用该模板的类型为集合（实现了 `IEnumerable` 接口）的目标元素，在调用 `HtmlHelper` 或者 `HtmlHelper<TModel>` 以显示或者编辑模式对其进行呈现的时候会遍历其中的每个元素，并根据基于集合元素的 Model 元数据决定对其的呈现方法。同样以我们定义的数据类型 `DemoModel` 为例，按照如下的方式将它的 `Foo` 属性类型改为对象数组，上面应用了 `UIHintAttribute` 特性并将模板名称设置为“Collection”。

```
public class DemoModel
{
    [UIHint("Collection")]
    public object[] Foo { get; set; }
}
```

然后我们按照如下的方式创建一个包含三个对象的数组，作为数据元素的三个对象类型分别是数字、字符串和布尔，然后将该数组作为 `Foo` 属性创建一个具体的 Model 对象。

```
object[] foo = new object[]
{
    123.00,
    "dummy text ...",
    true
};
DemoModel model = new DemoModel { Foo = foo };
```

在一个基于 Model 类型的强类型 View 中，我们分别调用 `HtmlHelper<TModel>` 的 `DisplayFor` 和 `EditorFor` 方法将上面创建的 `DemoModel` 对象的 `Foo` 属性以显示和编辑的模式呈现出来。

```
@model DemoModel
@Html.DisplayFor(m=>m.Foo)
@Html.EditorFor(m=>m.Foo)
```

Model 对象的 `Foo` 属性最终呈现出来的 HTML 如下所示，我们可以看到不论是显示模式还是编辑模式，基本上就是对集合元素呈现的 HTML 的组合而已。

```
123dummy text ...<input checked="checked" class="check-box" disabled=
"disabled" type="checkbox" />

<input class="text-box single-line" data-val="true" data-val-number="The
field Double must be a number." data-val-required="The Double field is
required." id="Foo_0_" name="Foo[0]" type="text" value="123" />

<input class="text-box single-line" id="Foo_1_" name="Foo[1]" type="text"
value="dummy text ..." />

<input checked="checked" class="check-box" data-val="true"
data-val-required="The Boolean field is required." id="Foo_2_" name="Foo[2]"
type="checkbox" value="true" />

<input name="Foo[2]" type="hidden" value="false" />
```

Object

我们说过 ASP.NET 内部采用基于类型的模板匹配策略，如果通过 ModelMetadata 对象表示的 Model 元数据不能找到一个具体的模板，最终都会落到 Object 模板上。Object 模板对目标对象的呈现方式很简单，它通过 ModelMetadata 的 Properties 属性得到所有基于属性的 Model 元数据。针对每个表示属性 Model 元数据的 ModelMetadata，它会根据 DisplayName，或者属性名称生成一个标签（实际上是一个内部文本为显示名称的<div>元素），然后根据元数据将属性值以显示或者编辑的模式呈现出来。

```
public class Address
{
    [DisplayName("省")]
    public string Province { get; set; }
    [DisplayName("市")]
    public string City { get; set; }
    [DisplayName("区")]
    public string District { get; set; }
    [DisplayName("街道")]
    public string Street { get; set; }
}
```

创建一个具体的 Address 对象并将其作为 Model 呈现在一个具有如下定义的 View 中，这是一个 Model 类型为 Address 的强类型 View，可以调用 HtmlHelper<TModel>的 DisplayForModel 方法将 Model 对象以显示模式呈现出来。

```
@model Address
@Html.DisplayForModel()
```

从如下所示的 HTML 中可以看出，作为 Model 的 Address 对象的所有属性都以显示模式呈现出来，而且在前面还具有相应的标签。

```
<div class="display-label">省</div>
<div class="display-field">江苏省</div>
<div class="display-label">市</div>
<div class="display-field">苏州市</div>
<div class="display-label">区</div>
<div class="display-field">工业园区</div>
<div class="display-label">街道</div>
<div class="display-field">星湖街 328 号</div>
```

值得一提的是，Object 模板在对属性进行遍历的过程中，不论是显示模式还是编辑模式，只会处理非复杂类型属性成员，也就是如果属性成员是一个复杂类型（不能支持针对字符串类型的转换），它不会出现在最终生成的 HTML 中。

```
public class Contact
{
    [DisplayName("姓名")]
    public string Name { get; set; }
    [DisplayName("电话")]
```

```

public string PhoneNo { get; set; }
[DisplayName("Email 地址")]
public string EmailAddress { get; set; }
[DisplayName("联系地址")]
public Address Address { get; set; }
}

```

通过上面的代码片段我们定义了一个表示联系人的数据类型 `Contact`，它具有一个类型为 `Address` 的同名属性。现在我们创建一个具体的 `Contact` 对象，并对包括 `Address` 属性在内的所有属性进行初始化，然后通过如下的方式调用 `HtmlHelper<TModel>` 的 `DisplayForModel` 方法将它呈现在以此作为 `Model` 的 `View` 中。

```

@model Contact
@Html.DisplayForModel()

```

从如下所示的 HTML 中可以看出，由于 `Contact` 的数据成员 `Address` 是复杂类型，其内容并不会呈现出来。

```

<div class="display-label">姓名</div>
<div class="display-field">张三</div>
<div class="display-label">电话</div>
<div class="display-field">1234567890</div>
<div class="display-label">Email 地址</div>
<div class="display-field">zhangsan@gmail.com</div>

```

可以有两种方式解决这个问题，一种是为 `Address` 类型定义相应的模板，另一种就是按照类似如下的方式手工将复杂类型属性成员呈现出来。

```

@model Contact
@Html.DisplayForModel()
@Html.EditorFor(m=>m.Address)

```

4.2.3 DataTypeName 与模板名称

我们知道，通过 `DataTypeAttribute` 特性为目标元素设置的数据类型最终会反映在 `ModelMetadata` 对象的 `DataTypeName` 属性上，对于某些的数据类型（如 `Date`、`Time`、`Currency` 等）还会创建一个相应的 `DisplayFormatAttribute` 特性应用到 `ModelMetadata` 上，那么 `ModelMetadata` 的 `DataTypeName` 属性对目标元素在 HTML 中的最终呈现具有怎样的影响呢？

实际上在模板匹配的过程中会将 `ModelMetadata` 的 `DataTypeName` 属性当作模板名称来看待，所以下面两种形式的 `DemoModel` 类型定义可以看成是等效的。通过 `UIHintAttribute` 特性设置的模板名称和通过 `DataTypeAttribute` 特性设置的数据类型的唯一不同之处在于前者具有更高的优先级。换句话说，如果将 `UIHintAttribute` 和 `DataTypeAttribute` 同时应用到同一个数据成员分别将模板名称和数据类型设置为“ABC”和“XYZ”，自定义模板 XYZ 只有在模板 ABC 不存在的情况下才会被使用。

```

public class DemoModel
{

```



```

        [DataType(DataType.Html)]
        public string Foo { get; set; }

        [DataType(DataType.MultilineText)]
        public string Bar { get; set; }

        [DataType(DataType.Url)]
        public string Baz { get; set; }
    }

    public class DemoModel
    {
        [UIHint("Html")]
        public string Foo { get; set; }

        [UIHint("MultilineText")]
        public string Bar { get; set; }

        [UIHint("Url")]
        public string Baz { get; set; }
    }

```

实例演示：证明 DataTypeName 与模板名称的等效性（S410）

为了证明通过 `DataTypeAttribute` 特性设置的数据类型在针对目标元素的可视化呈现过程中被视为模板名称，我们来做一个简单的实例演示。在这个实例中定义了如下一个表示三角形的数据类型 `Triangle`，其属性 `A`、`B` 和 `C` 是一个 `Point` 对象，表示三个角所在的坐标。

```

public class Triangle
{
    [DataType("PointInfo")]
    public Point A { get; set; }

    [DataType("PointInfo")]
    public Point B { get; set; }

    [DataType("PointInfo")]
    public Point C { get; set; }
}

[TypeConverter(typeof(PointTypeConverter))]
public class Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public Point(double x, double y)
    {
        this.X = x;
        this.Y = y;
    }

    public static Point Parse(string point)
    {
        string[] split = point.Split(',');
        if (split.Length != 2)
        {
            throw new FormatException("Invalid point expression.");
        }
    }
}

```

```

    }
    double x;
    double y;
    if (!double.TryParse(split[0], out x) ||
        !double.TryParse(split[1], out y))
    {
        throw new FormatException("Invalid point expression.");
    }
    return new Point(x, y);
}
}

public class PointTypeConverter : TypeConverter
{
    public override bool CanConvertFrom(ITypeDescriptorContext
        context, Type sourceType)
    {
        return sourceType == typeof(string);
    }

    public override object ConvertFrom(ITypeDescriptorContext
        context, CultureInfo culture, object value)
    {
        if (value is string)
        {
            return Point.Parse(value as string);
        }
        return base.ConvertFrom(context, culture, value);
    }
}

```

对于类型 `Triangle` 和 `Point` 的定义，有两点值得注意：第一，`Triangle` 的三个 `A`、`B` 和 `C` 属性上应用了 `DataTypeAttribute` 特性并将自定义数据类型设置为 `PointInfo`（不是 `Point`）；第二，`Point` 类型上应用了 `TypeConverterAttribute` 特性并将 `TypeConverter` 类型设置为 `PointTypeConverter`，后者支持源自字符串的类型转换。通过前面对复杂类型（Complex Type）的介绍，这样会将 `Triangle` 的三个属性从复杂类型成员转换成简单类型成员。根据前面介绍的关于 `Object` 模板对数据成员的遍历规则，`Triangle` 的这三个属性才能被最终呈现出来。

现在我们创建一个 `Model` 类型为 `Point` 的强类型 `View`，用它作为用于呈现 `Point` 对象的模板，并将其命名为 `PointInfo`（和前面通过 `DataTypeAttribute` 特性指定的自定义数据类型一致）。因为只需要为 `Point` 定义针对显示模式的模板，所以将具有如下定义的模板文件放在“`~/Views/Shared/DisplayTemplates`”目录下。如下面的代码片段所示，将一个 `Point` 对象显示为 `(X,Y)` 的形式。

```

@model Point
(@Model.X, @Model.Y)

```

现在我们创建如下一个 `HomeController`，在默认的 `Index` 操作方法中创建了一个 `Triangle` 对象将其呈现在默认的 `View` 中。

```

public class HomeController : Controller
{
    public ActionResult Index()

```

```

    {
        Triangle triangle = new Triangle
        {
            A = new Point(1,2),
            B = new Point (2,3),
            C = new Point(3,4)
        };
        return View(triangle);
    }
}

```

下面是 Action 方法 Index 对应 View 的定义，可以看出这是一个 Model 类型为 Triangle 的强类型 View。在该 View 中，我们仅仅调用了 HtmlHelper<TModel> 的扩展方法 DisplayModel 作为 Model 的 Triangle 对象以显示模式呈现出来。

```

@model Triangle
<html>
    <head>
        <title>Triangle</title>
    </head>
    <body>
        @Html.DisplayForModel()
    </body>
</html>

```

运行该 Web 应用后会在浏览器中得到如图 4-13 所示的呈现效果，可以看到 Triangle 对象的 A、B 和 C 属性表示的三个角的坐标是完全按照我们定义的 PointInfo 模板的方式来显示的。

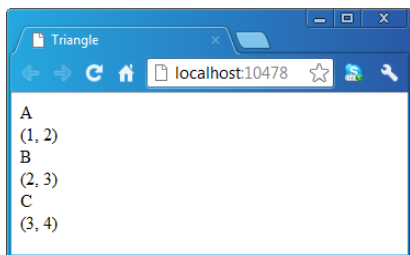


图 4-13 通过 DataTypeAttribute 特性设置模板名称的呈现效果

4.2.4 模板的获取与执行

当我们调用 HtmlHelper 或者 HtmlHelper<TModel> 的模板方法对整个 Model 或者 Model 的某个数据成员以某种模式（显示模式或者编辑模式）进行 HTML 呈现的时候，ASP.NET MVC 会根据预先创建的用于描述目标数据的 Model 元数据获取相应的模板。如果模板对应着某个自定义的 View，那么只需要执行该 View 即可；对于默认模板，则直接可以得到相应的 HTML。

根据 Model 元数据对模板的提取是整个模板方法执行流程中最核心的部分，也是本节讨论的重点。我们以 HtmlHelper<TModel> 的扩展方法 DisplayFor（定义如下）为例，看看针对通过表达式 expression 获取的 Model 对象是如何以显示模式呈现出来的。

```
public static class DisplayExtensions
{
    public static MvcHtmlString DisplayFor<TModel, TValue>(
        this HtmlHelper<TModel> html,
        Expression<Func<TModel, TValue>> expression, string templateName);
}
```

在 `DisplayFor` 被调用的时候，如果通过参数 `expression` 表示的 Model 获取表达式是针对某个属性的，那么属性名会被获取出来，然后执行表达式得到作为 Model 的对象，该对象连同属性名（如果有）一起被用于表示 Model 元数据的 `ModelMetadata` 对象的创建。接下来 ASP.NET MVC 会根据这个 `ModelMetadata` 对象得到一系列表示分部模板 View 名称的列表，这些 View 名称按照优先级排列如下。

- 作为参数 `templateName` 传入的模板名称（如果不为空）。
- `ModelMetadata` 的 `TemplateHint` 属性值（如果不为空）。
- `ModelMetadata` 的 `DataTypeName` 属性值（如果不为空）。
- 如果 Model 对象的真实类型为非空值类型，该类型名作为模板 View 名；否则底层（`Underlying`）类型名作为模板 View 名（比如，对于 `int?` 类型则将 `Int32` 作为模板 View 名）。
- 如果 Model 对象的真实类型为非复杂类型，会选择 `String` 模板（由于非复杂类型能够实现与 `String` 类型之间的转换，所以可以转换成 `String` 进行呈现）。
- 在 Model 的声明类型为接口情况下，如果该接口继承自 `IEnumerable` 则采用 `Collection` 模板。
- 在 Model 的声明类型为接口情况下，会选择 `Object` 模板。
- 如果 Model 声明类型不是接口类型，按照其类型继承关系向上追溯直到 `Object` 类型，逐个将类型名称作为模板 View 名称。如果声明类型实现了 `IEnumerable` 接口，则将最后的 `Object` 替换成 `Collection`。

对于得到的这个列表，ASP.NET MVC 会按照先后顺序遍历所有的元素，并将它们作为模板名称根据呈现模式在指定的路径（显示模式和编辑模式分别为 `“/DisplayTemplates/{TemplateName}”` 和 `“/EditorTemplates/{TemplateName}”`）去寻找定义模板的 Partial View。如果存在，会直接执行该 View 来对目标元素进行呈现；如果不能找到自定义模板分部 Partial View，则根据该模板名称在默认的模板列表中查找。如果存在匹配的默认模板，则按照对应的默认模板来呈现目标元素；如果默认的模板列表中的名称均与指定的名称不匹配，会进入下一次迭代。

ASP.NET MVC 在内部类型 `TemplateHelpers` 中定义了如下一个 `GetViewNames` 方法，它根据 `ModelMetadata`（`metadata` 参数）和作为 `TemplateHint` 的模板名称列表（`templateHints` 参数）返回一个按照选择优先级排列的“候选模板名称”列表。字符串数组参数 `templateHints` 不仅仅包含 `ModelMetadata` 的 `TemplateHint` 属性，还包括在调用模板方法时手工指定的模板名称（排在 `TemplateHint` 属性之前）和 `ModelMetadata` 表示数据类型名称的 `DataTypeName`

属性（排在 `TemplateHint` 之后），上面我们提到的“数据类型名称作为模板名称”的具体实现就体现在这里。

```
internal static class TemplateHelpers
{
    //其他成员
    internal static IEnumerable<string> GetViewNames (ModelMetadata metadata,
        params string[] templateHints);
}
```

可以通过一个简单的实例来演示模板方法在执行过程中对模板的选择机制。在一个 ASP.NET MVC 应用中定义了如下一个数据类型 `DemoModel`，其中属性 `Foo` 和 `Bar` 是简单类型（`int` 和 `int?`），属性 `Baz` 是复杂类型，而属性 `Qux` 是一个集合类型。四个属性上均应用了 `UIHintAttribute` 和 `DataTypeAttribute` 特性并作了相同的设置。

```
public class DemoModel
{
    [UIHint("TemplateHint")]
    [DataType("DataTypeName")]
    public int Foo { get; set; }

    [UIHint("TemplateHint")]
    [DataType("DataTypeName")]
    public int? Bar { get; set; }

    [UIHint("TemplateHint")]
    [DataType("DataTypeName")]
    public Baz Baz { get; set; }

    [UIHint("TemplateHint")]
    [DataType("DataTypeName")]
    public IEnumerable<int> Qux { get; set; }
}
public class Baz
{}
```

然后我们创建了如下一个 `HomeController`。静态方法 `GetCandidateTemplates` 根据 `ModelMetadata` 和显示指定的模板名称返回一个作为候选模板名称的字符串列表。由于我们需要使用到 `TemplateHelpers` 这个内部类型的 `GetViewNames` 方法，所以我们采用反射的方式来调用它。我们将包含显式指定模板名称、`ModelMetadata` 的 `TemplateHint` 属性和 `DataTypeName` 属性的字符串数组作为调用该方法的第二个参数。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ModelMetadata metadata = new ModelMetadata(
            ModelMetadataProviders.Current, null, null, typeof(DemoModel), null);
        ViewBag.TemplateNamesAccessor = new Func<ModelMetadata, string,
            IEnumerable<string>>(GetCandidateTemplates);
        return View(metadata.Properties);
    }

    static IEnumerable<string> GetCandidateTemplates (ModelMetadata modelMetadata,
```

```

        string template)
    {
        Type templateHelpers = Type.GetType("System.Web.Mvc.Html.TemplateHelpers,
            System.Web.Mvc, Version=4.0.0.0, Culture=neutral,
            PublicKeyToken=31bf3856ad364e35");
        MethodInfo getViewNames = templateHelpers.GetMethod("GetViewNames",
            BindingFlags.NonPublic | BindingFlags.Static);
        string[] templates = new string[] { template, modelMetadata.TemplateHint,
            modelMetadata.DataTypeName };
        return (IEnumerable<string>)getViewNames.Invoke(null,
            new object[] { modelMetadata, templates });
    }
}

```

默认的 Action 方法 Index 中，根据类型 DemoModel 创建一个 ModelMetadata 对象，并将其 Properties 属性（描述 DemoModel 四个属性的 Model 元数据列表）作为 Model 将默认的 View 呈现出来。在进行 View 呈现之前，我们设置了 ViewBag 的 TemplateNamesAccessor 属性，而属性值是一个基于 GetCandidateTemplates 方法的委托对象。

如下所示的是 Action 方法中 Index 对象的 View 的定义，这是一个基于 ModelMetadata 集合的强类型 View，在该 View 中，借助从 ViewBag 获取的委托对象将针对 DemoModel 四个属性的候选模板名称列表呈现出来。

```

@model IEnumerable<ModelMetadata>
<html>
    <head>
        <title>候选模板列表</title>
    </head>
    <body>
        @{
            Func<ModelMetadata, string, IEnumerable<string>>
                templateNamesAccessor = ViewBag.TemplateNamesAccessor;
        }
        <ul>
            @foreach (ModelMetadata metadata in Model)
            {
                <li>
                    <span>@metadata.PropertyName</span>
                    <ul>
                        @foreach (string templateName in
                            templateNamesAccessor(metadata, "Mandatory Template"))
                        {
                            <li>@templateName</li>
                        }
                    </ul>
                </li>
            }
        </ul>
    </body>
</html>

```

这个程序运行之后会在浏览器中呈现出如图 4-14 所示的输出结果。前面我们介绍的在模板方法执行过程中针对 Model 元数据的模板选择机制在这里得到了很好的体现。针对 DemoModel 的每一个属性，按照解析出来的模板名称顺序，如果在预定义的目录下存在相应的模板 View 或者默认模板，它们将被用于该属性值的最终呈现。（S411）

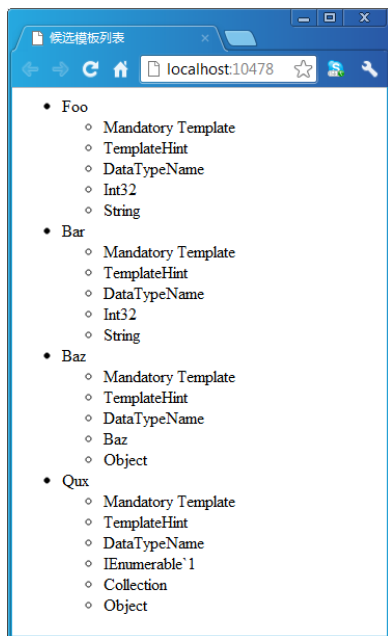


图 4-14 根据 Model 元数据得到的“候选模板名称”列表

4.2.5 实例演示：通过定制 Model 元数据和自定义模板实现预定义列表的呈现（S412）

根据 Model 元数据，我们不仅可以创建相应的模板来控制某种类型的数据在 UI 界面上的呈现方式，还可以通过一些扩展来控制 Model 元数据本身。在某些情况下通过这两者的结合往往可以解决很多特殊数据的呈现问题，接下来演示的实例就是一个典型的例子。

传统的 ASP.NET 具有一组重要的控件类型叫做列表控件（ListControl），它的子类包括 DropDownList、ListBox、RadioButtonList 和 CheckBoxList 等。对于 ASP.NET MVC 来说，可以通过 `HtmlHelper/HtmlHelper<TModel>` 的扩展方法 `DropDownList/DropDownListFor` 和 `ListBox/ListBoxFor` 在界面上呈现一个下拉框和列表框，但是需要手工指定包含的所有列表选项。在一般的 Web 应用中，尤其是企业应用中，我们会选择将这些列表进行单独地维护。如果我们在构建“列表控件”的时候能够免去手工提供列表的工作，这无疑会为开发带来极大的便利，而实际上这很容易实现。

我们先来看看通过该扩展最终实现的效果。我们在 ASP.NET MVC 应用中定义一个代表员工的 `Employee` 类型。如下面的代码片段所示，表示性别、学历、部门和技能的属性分别应用了 `RadioButtonListAttribute`、`DropDownListAttribute`、`ListBoxAttribute` 和 `CheckBoxListAttribute` 四个特性。从名称可以看出来，这四个特性分别代表了目标元素呈现在 UI 界面上的形式，即对应着传统 ASP.NET Web 应用中的四种类型的列表控件：`RadioButtonList`、`DropDownList`、

ListBox 和 CheckBoxList, 特性中指定的字符串表示预定义列表的名称。

```
public class Employee
{
    [DisplayName("姓名")]
    public string Name { get; set; }

    [RadioButtonList("Gender")]
    [DisplayName("性别")]
    public string Gender { get; set; }

    [DropDownList("Education")]
    [DisplayName("学历")]
    public string Education { get; set; }

    [ListBox("Department")]
    [DisplayName("所在部门")]
    public IEnumerable<string> Departments { get; set; }

    [CheckBoxList("Skill")]
    [DisplayName("擅长技能")]
    public IEnumerable<string> Skills { get; set; }
}
```

在创建的默认 HomeController 中, 我们定义了如下一个 Index 操作方法, 在该方法中, 我们创建了一个具体的 Employee 对象并对它的所有属性进行了相应设置, 最终将该对象呈现在默认的 View 中。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        Employee employee = new Employee
        {
            Name      = "张三",
            Gender     = "M", //男
            Education  = "M", //硕士
            Departments = new string[] { "HR", "AD" }, //人事部, 行政部
            Skills     = new string[] { "CSharp", "AdoNet" } //C#, ADO.NET
        };
        return View(employee);
    }
}
```

如下所示的是 Action 方法中 Index 所对应的 View 的定义, 这是一个基于 Employee 的强类型 View。在该 View 中, 通过调用 HtmlHelper<TModel>的模板方法 EditorFor 将作为 Model 的 Employee 对象的所有属性以编辑模式呈现出来。

```
@model Employee
<html>
<head>
    <title>编辑员工信息</title>
</head>
<body>
```



```

<table>
  <tr><td>@Html.LabelFor(m => m.Name)</td><td>
    @Html.EditorFor(m => m.Name)</td></tr>
  <tr><td>@Html.LabelFor(m => m.Gender)</td><td>
    @Html.EditorFor(m => m.Gender)</td></tr>
  <tr><td>@Html.LabelFor(m => m.Education)</td><td>
    @Html.EditorFor(m => m.Education)</td></tr>
  <tr><td>@Html.LabelFor(m => m.Departments)</td><td>
    @Html.EditorFor(m => m.Departments)</td></tr>
  <tr><td>@Html.LabelFor(m => m.Skills)</td><td>
    @Html.EditorFor(m => m.Skills)</td></tr>
</table>
</body>
</html>

```

图 4-15 体现了该 Web 应用运行时的效果，可以看到，四个属性分别以四种不同的“列表控件”呈现出来，并与应用在它们上面的四个自定义的列表特性（RadioButtonListAttribute、DropDownListAttribute、ListBoxAttribute 和 CheckBoxListAttribute）相匹配。

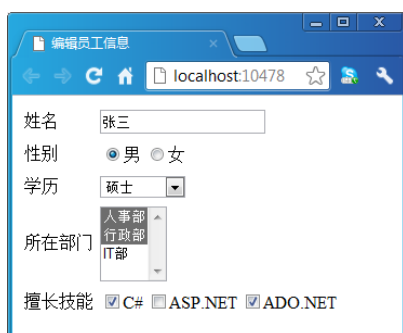


图 4-15 应用 ListAttribute 特性的数据成员的呈现效果

现在我们对上面演示的针对自动化“列表控件”呈现的设计原理进行简单介绍。首先定义了如下一个表示列表中某个条目（列表项）的类型 `ListItem`，简单起见，仅仅定义 `Text` 和 `Value` 两个属性，它们分别表示显示的文字和代表的值。对于一组表示国家的列表，列表项的 `Text` 属性表示成国家名称（比如“中国”），具体的值则可能是国家的代码（比如“CN”）。

```

public class ListItem
{
    public string Text { get; set; }
    public string Value { get; set; }
}

```

将提供列表数据的组件称为 `ListProvider`，它们实现了 `IListProvider` 接口。如下面的代码片段所示，`IListProvider` 具有唯一的方法 `GetListItems`，根据指定的列表名称获取所有的列表项。通过实现 `IListProvider`，我们定义了一个默认的 `DefaultListProvider`。简单起见，`DefaultListProvider` 直接通过一个静态字段模拟列表的存储，在真正的项目中一般会保存在数据库中。`DefaultListProvider` 维护了四组列表，分别表示“性别”、“学历”、“部门”和“技能”，它们正好对应着 `Employee` 的四个属性。

```

public interface IListProvider
{
    IEnumerable<ListItem> GetListItems(string listName);
}

public class DefaultListProvider : IListProvider
{
    private static Dictionary<string, IEnumerable<ListItem>> listItems
        = new Dictionary<string, IEnumerable<ListItem>>();
    static DefaultListProvider()
    {
        var items = new ListItem[]{
            new ListItem{ Text = "男", Value="M"},
            new ListItem{ Text = "女", Value="F"};
        listItems.Add("Gender", items);

        items = new ListItem[]{
            new ListItem{ Text = "高中", Value="H"},
            new ListItem{ Text = "大学本科", Value="B"},
            new ListItem{ Text = "硕士", Value="M"},
            new ListItem{ Text = "博士", Value="D"};
        listItems.Add("Education", items);

        items = new ListItem[]{
            new ListItem{ Text = "人事部", Value="HR"},
            new ListItem{ Text = "行政部", Value="AD"},
            new ListItem{ Text = "IT 部", Value="IT"};
        listItems.Add("Department", items);

        items = new ListItem[]{
            new ListItem{ Text = "C#", Value="CSharp"},
            new ListItem{ Text = "ASP.NET", Value="AspNet"},
            new ListItem{ Text = "ADO.NET", Value="AdoNet"};
        listItems.Add("Skill", items);
    }

    public IEnumerable<ListItem> GetListItems(string listName)
    {
        IEnumerable<ListItem> items;
        if (listItems.TryGetValue(listName, out items))
        {
            return items;
        }
        return new ListItem[0];
    }
}

```

接下来定义如下一个 `ListProviders` 类型，它的静态只读属性 `Current` 表示当前的 `IListProvider`，而对当前 `IListProvider` 的注册通过静态方法 `SetListProvider` 来实现。如果没有对当前 `IListProvider` 进行显式注册，则默认采用 `DefaultListProvider`。

```

public static class ListProviders
{
    public static IListProvider Current { get; private set; }

    static ListProviders()

```

```

    {
        Current = new DefaultListProvider();
    }

    public static void SetListProvider(Func<IListProvider> providerAccessor)
    {
        Current = providerAccessor();
    }
}

```

基于四种“列表控件”的 HTML 的生成是通过定义 `HtmlHelper` 的扩展方法来实现的，如下面的代码所示，定义在 `ListControlExtensions` 中的四个扩展方法实现了针对这四种列表控件的 UI 呈现。参数 `listName` 表示使用的预定义列表的名称，而 `value` 和 `values` 则表示绑定的值。`RadioButtonList/DropDownList` 只允许单项选择，而 `ListBox/CheckBoxList` 允许多项选择，所以对应的值类型分别是 `string` 和 `IEnumerable<string>`。

```

public static class ListControlExtensions
{
    //其他成员
    public static MvcHtmlString RadioButtonList(
        this HtmlHelper htmlHelper, string name, string listName,
        string value)
    {
        return RadioButtonCheckBoxList(htmlHelper, listName, item =>
            htmlHelper.RadioButton(name, item.Value, value == item.Value));
    }

    public static MvcHtmlString CheckBoxList(this HtmlHelper htmlHelper,
        string name, string listName, IEnumerable<string> values)
    {
        return RadioButtonCheckBoxList(htmlHelper, listName,
            item => CheckBoxWithValue(htmlHelper, name,
                values.Contains(item.Value), item.Value));
    }

    public static MvcHtmlString ListBox(this HtmlHelper htmlHelper,
        string name, string listName, IEnumerable<string> values)
    {
        var listItems = ListProviders.Current.GetListItems(listName);
        List<SelectListItem> selectListItems = new List<SelectListItem>();
        foreach (var item in listItems)
        {
            selectListItems.Add(new SelectListItem { Value = item.Value,
                Text = item.Text,
                Selected = values.Any(value => value == item.Value) });
        }
        return htmlHelper.ListBox(name, selectListItems);
    }

    public static MvcHtmlString DropDownList(this HtmlHelper htmlHelper,
        string name, string listName, string value)
    {
        var listItems = ListProviders.Current.GetListItems(listName);
        List<SelectListItem> selectListItems = new List<SelectListItem>();
        foreach (var item in listItems)
        {

```

```

        selectListItems.Add(new SelectListItem { Value = item.Value,
            Text = item.Text, Selected = value == item.Value});
    }
    return htmlHelper.DropDownList(name, selectListItems);
}
}

```

从上面的代码片段可以看到,在 `ListBox` 和 `DropDownList` 方法中通过当前的 `ListProvider` 获取指定列表名称的所有列表项并生成相应的 `SelectListItem` 列表,最终通过调用 `HtmlHelper` 现有的扩展方法 `ListBox` 和 `DropDownList` 实现 HTML 的呈现。而 `RadioButtonList` 和 `CheckBoxList` 最终调用了辅助方法 `RadioButtonCheckBoxList` 显示了最终的 HTML 生成,该方法定义如下。

```

public static class ListControlExtensions
{
    public static MvcHtmlString CheckBoxWithValue(
        this HtmlHelper htmlHelper, string name, bool isChecked, string value)
    {
        string fullHtmlFieldName = htmlHelper.ViewContext.ViewData
            .TemplateInfo.GetFullHtmlFieldName(name);
        ModelState modelState;

        //将 ModelState 设置为表示是否勾选布尔值
        if (htmlHelper.ViewData.ModelState.TryGetValue(fullHtmlFieldName,
            out modelState))
        {
            htmlHelper.ViewData.ModelState.SetModelValue(fullHtmlFieldName,
                new ValueProviderResult(isChecked, isChecked.ToString(),
                    CultureInfo.CurrentCulture));
        }
        MvcHtmlString html;
        try
        {
            html = htmlHelper.CheckBox(name, isChecked);
        }
        finally
        {
            //将 ModelState 还原
            if (null != modelState)
            {
                htmlHelper.ViewData.ModelState[fullHtmlFieldName] = modelState;
            }
        }
        string htmlString = html.ToHtmlString();
        var index = htmlString.LastIndexOf('<');
        //过滤掉类型为"hidden"的<input>元素
        XElement element = XElement.Parse(htmlString.Substring(0, index));
        element.SetAttributeValue("value", value);
        return new MvcHtmlString(element.ToString());
    }

    private static MvcHtmlString RadioButtonCheckBoxList(
        HtmlHelper htmlHelper, string listName,
        Func<ListItem, MvcHtmlString> elementHtmlAccessor)
    {
        var listItems = ListProviders.Current.GetListItems(listName);
    }
}

```

```

        TagBuilder table = new TagBuilder("table");
        TagBuilder tr = new TagBuilder("tr");
        foreach (var listItem in listItems)
        {
            TagBuilder td = new TagBuilder("td");
            td.InnerHtml += elementHtmlAccessor(listItem).ToHtmlString();
            td.InnerHtml += listItem.Text;
            tr.InnerHtml += td.ToString();
        }
        table.InnerHtml = tr.ToString();
        return new MvcHtmlString(table.ToString());
    }
}

```

方法 `RadioButtonCheckBoxList` 在生成 `RadioButtonList` 和 `CheckBoxList` 的时候使用 `<table>` 进行布局。组成 `RadioButtonList` 的单个 `RadioButton` 最终是调用 `HtmlHelper` 现有的扩展方法 `RadioButton` 生成的，而 `CheckBoxList` 中的 `CheckBox` 则是通过调用我们自定义的 `CheckBoxWithValue` 方法生成的。`CheckBoxWithValue` 最终还是调用 `HtmlHelper` 现有的扩展方法 `CheckBox` 生成单个 `CheckBox` 对应的 HTML。但是该方法支持布尔值的绑定，并且会生成一个在这里不需要的 `Hidden` 元素，所以不得不在调用该方法的前后“作一些手脚”。

现在来介绍应用在 `Employee` 属性上的四个特性的定义。如下面的代码片段所示，基于四种“列表控件”的特性均继承自抽象特性 `ListAttribute`。`ListAttribute` 实现了 `IMetadataAware` 接口，在实现的 `OnMetadataCreated` 方法中将代表列表名称的 `ListName` 属性添加到 `ModelMetadata` 对象的 `AdditionalValues` 属性中。四个具体的列表特性重写了 `OnMetadataCreated` 方法，并在此基础上将 `ModelMetadata` 的 `TemplateHint` 分别设置为 `DropDownList`、`ListBox`、`RadioButtonList` 和 `CheckBoxList`。

```

[AttributeUsage(AttributeTargets.Property)]
public abstract class ListAttribute : Attribute, IMetadataAware
{
    public string ListName { get; private set; }
    public ListAttribute(string listName)
    {
        this.ListName = listName;
    }
    public virtual void OnMetadataCreated(ModelMetadata metadata)
    {
        metadata.AdditionalValues.Add("ListName", this.ListName);
    }
}

[AttributeUsage(AttributeTargets.Property)]
public class DropDownListAttribute : ListAttribute
{
    public DropDownListAttribute(string listName)
        : base(listName)
    {
    }
    public override void OnMetadataCreated(ModelMetadata metadata)
    {
        base.OnMetadataCreated(metadata);
        metadata.TemplateHint = "DropDownList";
    }
}

```

```

}

[AttributeUsage(AttributeTargets.Property)]
public class ListBoxAttribute : ListAttribute
{
    public ListBoxAttribute(string listName)
        : base(listName)
    { }
    public override void OnMetadataCreated(ModelMetadata metadata)
    {
        base.OnMetadataCreated(metadata);
        metadata.TemplateHint = "ListBox";
    }
}

[AttributeUsage(AttributeTargets.Property)]
public class RadioButtonListAttribute : ListAttribute
{
    public RadioButtonListAttribute(string listName)
        : base(listName)
    { }

    public override void OnMetadataCreated(ModelMetadata metadata)
    {
        base.OnMetadataCreated(metadata);
        metadata.TemplateHint = "RadioButtonList";
    }
}

[AttributeUsage(AttributeTargets.Property)]
public class CheckBoxListAttribute : ListAttribute
{
    public CheckBoxListAttribute(string listName)
        : base(listName)
    { }

    public override void OnMetadataCreated(ModelMetadata metadata)
    {
        base.OnMetadataCreated(metadata);
        metadata.TemplateHint = "CheckBoxList";
    }
}

```

由于四个具体的 `ListAttribute` 已经对表示模板名称的 `ModelMetadata` 的 `TemplateHint` 进行了设置，如果针对它们定义相应的 `Partial View` 作为对应的模板，那么在调用 `HtmlHelper/HtmlHelper<TModel>` 相应模板方法的时候就会按照这些模板对目标元素进行呈现。实现如图 4-15 所示的效果的四个模板定义如下，它们被保存在“~/View/Shared/EditorTemplates”目录下。

```

DropDownList.cshtml:
@model string
@{
    string listName =
        (string) ViewData.ModelMetadata.AdditionalValues["ListName"];
    @Html.DropDownList("", listName, Model)
}

```

```

ListBox.cshtml:
@model IEnumerable<string>
@{
    string listName =
        (string) ViewData.ModelMetadata.AdditionalValues["ListName"];
    @Html.ListBox("", listName, Model)
}

RadioButtonList.cshtml:
@model string
@{
    string listName =
        (string) ViewData.ModelMetadata.AdditionalValues["ListName"];
    @Html.RadioButtonList("", listName, Model)
}

CheckBoxList.cshtml:
@model IEnumerable<string>
@{
    string listName =
        (string) ViewData.ModelMetadata.AdditionalValues["ListName"];
    @Html.CheckBoxList("", listName, Model)
}

```

4.3 Model 元数据的提供机制

表示 Model 元数据的 `ModelMetadata` 对象最终是通过一个名为 `ModelMetadataProvider` 的组件提供的, 接下来我们着重讨论基于 `ModelMetadataProvider` 的 Model 元数据提供机制及其扩展。

4.3.1 再谈 ModelMetadata

我们在前面已经对用于描述 Model 元数据的 `ModelMetadata` 对象进行了非常详细的介绍, 但是它还具有有一些子类值得深入探讨。在之前介绍 `ModelMetadata` 的章节中, 我们主要讨论了它用于描述 Model 元数据的一些属性, 现在来简单看看它的构造函数。

`ModelMetadata` 唯一的构造函数如下面的代码片段所示。参数 `provider` 用于指定提供 Model 元数据的 `ModelMetadataProvider` 对象, 包含在 `Properties` 属性中针对属性的 Model 元数据都是通过它来提供的。如果创建针对属性的 Model 元数据, 可以通过参数 `propertyName` 和 `containerType` 指定属性名称和容器类型。参数 `modelType` 表示对应数据类型, 而 `modelAccessor` 是一个获取作为 Model 对象 (数据对象) 的委托。

```

public class ModelMetadata
{
    // 其他成员
    public ModelMetadata(ModelMetadataProvider provider, Type containerType,
        Func<object> modelAccessor, Type modelType, string propertyName);
}

```

DataAnnotationsModelMetadata

由于 ASP.NET MVC 采用了基于数据注解特性的声明式定义，所以 ASP.NET MVC 定义了一个名为 `System.Web.Mvc.DataAnnotationsModelMetadata` 的类型。如下面的代码片段所示，继承自 `ModelMetadata` 的 `DataAnnotationsModelMetadata` 重写了 `GetSimpleDisplayText` 方法，它的返回值作为 `ModelMetadata` 的 `SimpleDisplayText` 属性值，表示简单显示文本。

```
public class DataAnnotationsModelMetadata : ModelMetadata
{
    public DataAnnotationsModelMetadata(DataAnnotationsModelMetadataProvider
        provider, Type containerType, Func<Object> modelAccessor, Type modelType,
        string propertyName, DisplayColumnAttribute displayColumnAttribute);

    protected override string GetSimpleDisplayText();
}
```

`DataAnnotationsModelMetadata` 的构造函数在基类构造函数参数列表上提供了一个具有如下定义的 `DisplayColumnAttribute` 特性对象，它具有一个 `DisplayColumn` 属性用于指定作为显示文本的属性名称。如果该特性不为 `Null`，`GetSimpleDisplayText` 方法会先获取其 `DisplayColumn` 属性值，然后在 `Model` 类型中找到对应的属性并通过反射得到并返回具体的属性值。

```
[AttributeUsage(AttributeTargets.Class, Inherited=true, AllowMultiple=false)]
public class DisplayColumnAttribute : Attribute
{
    //其他成员
    public DisplayColumnAttribute(string displayColumn);
    public string DisplayColumn { get; }
}
```

以前面定义的数据类型 `Contact/Address` 为例，我们在 `Address` 类型上添加一个字符串类型的属性 `DisplayText`，而类型 `Address` 上应用了 `DisplayColumnAttribute` 特性并将其 `DisplayColumn` 属性设置为“`DisplayText`”。那么针对一个具体 `Contact` 对象的 `DataAnnotationsModelMetadata` 对象来说（其 `Contact` 对象作为其 `Model` 属性），针对属性 `Address` 的 `ModelMetadata` 的 `SimpleDisplayText` 属性就是 `Address` 的 `DisplayText` 属性值。

```
public class Contact
{
    //其他成员
    public Address Address { get; set; }
}

[DisplayColumn("DisplayText")]
public class Address
{
    //其他成员
    public string DisplayText { get; set; }
}
```

CachedDataAnnotationsModelMetadata

很多读者会认为 `DataAnnotationsModelMetadata` 是 ASP.NET MVC 默认使用的描述

Model 元数据的类型，实则不然。System.Web.Mvc.CachedDataAnnotationsModelMetadata 才是默认使用的 ModelMetadata 类型，而且 CachedDataAnnotationsModelMetadata 和 DataAnnotationsModelMetadata 没有任何关系。在介绍 CachedDataAnnotationsModelMetadata 之前，先来了解一下它的基类 System.Web.Mvc.CachedModelMetadata<TPrototypeCache>。

```
public abstract class CachedModelMetadata<TPrototypeCache> : ModelMetadata
{
    protected CachedModelMetadata(CachedModelMetadata<TPrototypeCache> prototype,
        Func<object> modelAccessor);
    protected CachedModelMetadata(
        CachedDataAnnotationsModelMetadataProvider provider, Type containerType,
        Type modelType, string propertyName, TPrototypeCache prototypeCache);

    protected virtual string ComputeDataTypeName();
    //其他 ComputeXxx 方法

    protected TPrototypeCache          PrototypeCache { get; set; }
    public sealed override string      DataTypeName { get; set; }
    //其他 Model 元数据属性
}
```

CachedModelMetadata<TPrototypeCache>采用了类似于“原型”的设计，它的一个构造函数中会指定另一个作为原型的 CachedModelMetadata<TPrototypeCache> 对象，而 PrototypeCache 属性就来源这个原型对象的同名属性。另一个构造函数接受一个用于创建 CachedDataAnnotationsModelMetadata 的 CachedDataAnnotationsModelMetadataProvider 对象（笔者个人觉得这违反了“依赖倒置”的设计原则），通过它指定用于初始化 PrototypeCache 属性的参数 prototypeCache。

CachedModelMetadata<TPrototypeCache>重写了所有描述 Model 元数据的属性，并且定义相应的受保护虚方法 ComputeXxx 来最终“计算”它们的属性值。在上面给出的代码片段中，我们仅仅列出了重写的 DataTypeName 属性和对应的 ComputeDataTypeName 方法。ComputeDataTypeName 方法仅仅会执行一次，而计算结果会通过字段的形式保存起来，具体的逻辑反映在如下所示的代码片段中。

```
public abstract class CachedModelMetadata<TPrototypeCache> : ModelMetadata
{
    //其他成员
    private string _dataTypeName;
    private bool _dataTypeNameComputed;

    protected virtual string ComputeDataTypeName()
    {
        return base.DataTypeName;
    }
    public sealed override string DataTypeName
    {
        get
        {
            return CachedModelMetadata<TPrototypeCache>.CacheOrCompute<string>(
                new Func<string>(this.ComputeDataTypeName),
                ref this._dataTypeName, ref this._dataTypeNameComputed);
        }
    }
}
```

```

    }
    set
    {
        this._dataTypeName = value;
        this._dataTypeNameComputed = true;
    }
}
private static TResult CacheOrCompute<TResult>(Func<TResult> computeThunk,
    ref TResult value, ref bool computed)
{
    if (!computed)
    {
        value = computeThunk();
        computed = true;
    }
    return value;
}
}

```

从上面的代码片段可以看出，用于计算某个 Model 元数据属性值的 `ComputeXxx` 方法（`ComputeDataTypeName`）直接返回基类的对应的属性值，`CachedDataAnnotationsModelMetadata` 会重写这个方法。如下面的代码片段所示，`CachedDataAnnotationsModelMetadata` 继承自 `CachedModelMetadata<CachedDataAnnotations MetadataAttributes>`，作为缓存 Model 元数据信息的类型 `System.Web.Mvc.Cached DataAnnotationsMetadataAttributes` 中包含了用于定义 Model 元数据的所有数据注解特性。

```

public class CachedDataAnnotationsModelMetadata :
    CachedModelMetadata<CachedDataAnnotationsMetadataAttributes>
{
    public CachedDataAnnotationsModelMetadata(
        CachedDataAnnotationsModelMetadata prototype,
        Func<object> modelAccessor);
    public CachedDataAnnotationsModelMetadata(
        CachedDataAnnotationsModelMetadataProvider provider, Type containerType,
        Type modelType, string propertyName, IEnumerable<Attribute> attributes);

    protected override string ComputeDataTypeName();
    //其他重写的 ComputeXxx 方法
}

public class CachedDataAnnotationsMetadataAttributes
{
    public CachedDataAnnotationsMetadataAttributes(Attribute[] attributes);

    public DataTypeAttribute           DataType { get; protected set; }
    public DisplayAttribute             Display { get; protected set; }
    public DisplayColumnAttribute       DisplayColumn { get; protected set; }
    public DisplayFormatAttribute       DisplayFormat { get; protected set; }
    public DisplayNameAttribute         DisplayName { get; protected set; }
    public EditableAttribute            Editable { get; protected set; }
    public HiddenInputAttribute         HiddenInput { get; protected set; }
    public ReadOnlyAttribute            ReadOnly { get; protected set; }
    public RequiredAttribute            Required { get; protected set; }
    public ScaffoldColumnAttribute      ScaffoldColumn { get; protected set; }
    public UIHintAttribute              UIHint { get; protected set; }
}

```

实际上用于计算 Model 元数据属性值的 `ComputeXxx` 方法直接通过包含在这个 `CachedDataAnnotationsMetadataAttributes` 对象中对应的特性获取相应的值。下面的代码片段体现了 `ComputeDataTypeName` 方法用于计算数据类型名称的逻辑。

```
public class CachedDataAnnotationsModelMetadata : ...
{
    //其他成员
    protected override string ComputeDataTypeName()
    {
        if (base.PrototypeCache.DataType != null)
        {
            return base.PrototypeCache.DataType.ToDataTypeName(null);
        }
        if ((base.PrototypeCache.DisplayFormat != null)
            && !base.PrototypeCache.DisplayFormat.HtmlEncode)
        {
            return DataType.Html.ToString();
        }
        return base.ComputeDataTypeName();
    }
}
```

4.3.2 ModelMetadataProvider

在 ASP.NET MVC 的 Model 元数据相关的应用编程接口中，用于创建 Model 元数据的 `ModelMetadataProvider` 继承自抽象类 `System.Web.Mvc.ModelMetadataProvider`。如下面的代码片段所示，`ModelMetadataProvide` 具有三个抽象方法：`GetMetadataForProperties` 方法用于获取表示针对指定容器对象和类型所有属性的 Model 元数据集；`GetMetadataForProperty` 获取针对指定容器对象和类型某个具体属性对象的 Model 元数据；而 `GetMetadataForType` 则直接返回针对容器对象和类型的 Model 元数据。

```
public abstract class ModelMetadataProvider
{
    public abstract IEnumerable<ModelMetadata> GetMetadataForProperties(
        object container, Type containerType);

    public abstract ModelMetadata GetMetadataForProperty(
        Func<object> modelAccessor, Type containerType, string propertyName);

    public abstract ModelMetadata GetMetadataForType(Func<object> modelAccessor,
        Type modelType);
}
```

AssociatedMetadataProvider

不论是用于创建 `DataAnnotationsModelMetadata` 的 `DataAnnotationsModelMetadataProvider`，还是用于创建 `CachedDataAnnotationsModelMetadata` 的 `CachedDataAnnotationsModelMetadataProvider`，它们都是 `System.Web.Mvc.AssociatedMetadataProvider` 的子类。`AssociatedMetadataProvider` 的主要作用是对应用在 Model 类型或属性上所有“关联”的特性

进行解析从而获取定义的 Model 元数据信息,这也是它命名的由来。如下面的代码片段所示, `AssociatedMetadataProvider` 实现了定义在 `ModelMetadataProvider` 的三个方法。

```
public abstract class AssociatedMetadataProvider : ModelMetadataProvider
{
    protected abstract ModelMetadata CreateMetadata(
        IEnumerable<Attribute> attributes, Type containerType,
        Func<object> modelAccessor, Type modelType, string propertyName);

    public override IEnumerable<ModelMetadata> GetMetadataForProperties(
        object container, Type containerType);
    public override ModelMetadata GetMetadataForProperty(
        Func<object> modelAccessor, Type containerType, string propertyName);
    public override ModelMetadata GetMetadataForType(Func<object> modelAccessor,
        Type modelType);
}
```

其实针对 `ModelMetadata` 的创建体现在抽象方法 `CreateMetadata` 上,它根据作为参数提供的特性列表得到相应的 Model 元数据信息。对于实现的定义在 `ModelMetadataProvider` 中的三个方法来说,它们仅仅是通过反射获取应用在 Model 类型和对应属性上的所有特性,并将这个特性列表作为参数传入抽象方法 `CreateMetadata`,以返回创建的 `ModelMetadata` 对象。

值得一提的是,在调用 `CreateMetadata` 创建出的 `ModelMetadata` 之后,ASP.NET MVC 会从特性列表中筛选出实现了 `IMetadataAware` 接口的特性,并将该 `ModelMetadata` 对象作为参数调用它们的 `OnMetadataCreated` 方法,所以实现了 `IMetadataAware` 接口的特性对 Model 元数据的定制具有最高的优先级。

DataAnnotationsModelMetadataProvider

`System.Web.Mvc.DataAnnotationsModelMetadataProvider` 针对于 `DataAnnotationsModelMetadata` 对象的创建。如下面的代码片段所示,它是 `AssociatedMetadataProvider` 的子类。在实现的 `CreateMetadata` 方法中,它会先从提供的所有特性列表中提取 `DisplayColumnAttribute` 特性并调用构造函数创建一个 `DataAnnotationsModelMetadata` 对象,然后获取用于定义 Model 元数据相关的数据注解特性并对其进行初始化。

```
public class DataAnnotationsModelMetadataProvider : AssociatedMetadataProvider
{
    public DataAnnotationsModelMetadataProvider();
    protected override ModelMetadata CreateMetadata(
        IEnumerable<Attribute> attributes, Type containerType,
        Func<object> modelAccessor, Type modelType, string propertyName);
}
```

CachedAssociatedMetadataProvider<TModelMetadata>

ASP.NET MVC 用于描述 Model 元数据的 `CachedDataAnnotationsModelMetadata` 是通过对应的 `CachedDataAnnotationsModelMetadataProvider` 提供的,而后者直接继承自具有如下定义的抽象类 `System.Web.Mvc.CachedAssociatedMetadataProvider<TModelMetadata>`,泛型参

数 TModelMetadata 继承自 ModelMetadata。

```
public abstract class CachedAssociatedMetadataProvider<TModelMetadata> :
AssociatedMetadataProvider where TModelMetadata: ModelMetadata
{
    protected sealed override ModelMetadata CreateMetadata(
        IEnumerable<Attribute> attributes, Type containerType,
        Func<object> modelAccessor, Type modelType, string propertyName);

    protected abstract TModelMetadata CreateMetadataFromPrototype(
        TModelMetadata prototype, Func<object> modelAccessor);
    protected abstract TModelMetadata CreateMetadataPrototype(
        IEnumerable<Attribute> attributes, Type containerType, Type modelType,
        string propertyName);

    protected CacheItemPolicy          CacheItemPolicy { get; set; }
    protected ObjectCache              PrototypeCache { get; set; }
}
```

基于对创建的 Model 元数据的缓存实现在 CachedAssociatedMetadataProvider<TModelMetadata>中。如上面的代码片段所示，它具有一个类型为 ObjectCache 的受保护属性 PrototypeCache，它实现了对针对某个类型或者定义在某个类型中某个属性所创建的 Model 元数据的缓存。

具体来说，在 CachedAssociatedMetadataProvider<TModelMetadata>中重写的 CreateMetadata 方法会根据 Model 类型和属性名称（针对基于属性的 Model 元数据创建）生成一个 Key，并借助 ObjectCache 对象从缓存中获取预先被缓存的 ModelMetadata 对象。如果存在着这么一个被缓存的 ModelMetadata 对象，ASP.NET MVC 会以此为原型调用抽象方法 CreateMetadataFromPrototype 创建并返回一个新的 ModelMetadata 对象，否则会调用另一个抽象方法 CreateMetadataPrototype 创建作为原型的 ModelMetadata 对象，该对象在被用于创建返回的 ModelMetadata 之前会被缓存。

默认情况下缓存的过期时间为 20 分钟，如果我们对包括过期时间在内的缓存策略进行定制，可以通过其受保护属性 CacheItemPolicy 来实现。

CachedDataAnnotationsModelMetadataProvider

System.Web.Mvc.CachedDataAnnotationsModelMetadataProvider 用于实现针对默认 Model 元数据类型 CachedDataAnnotationsModelMetadata 的提供。如下面的代码片段所示，它直接继承自 CachedAssociatedMetadataProvider<CachedDataAnnotationsModelMetadata>，实现的两个方法 CreateMetadataFromPrototype/CreateMetadataPrototype 直接创建并返回一个 CachedDataAnnotationsModelMetadata 对象。

```
public class CachedDataAnnotationsModelMetadataProvider :
    CachedAssociatedMetadataProvider<CachedDataAnnotationsModelMetadata>
{
    protected override CachedDataAnnotationsModelMetadata
        CreateMetadataFromPrototype(CachedDataAnnotationsModelMetadata prototype,
```

```

    Func<object> modelAccessor)
    {
        return new CachedDataAnnotationsModelMetadata(prototype, modelAccessor);
    }

    protected override CachedDataAnnotationsModelMetadata
        CreateMetadataPrototype(IEnumerable<Attribute> attributes,
            Type containerType, Type modelType, string propertyName)
    {
        return new CachedDataAnnotationsModelMetadata(this, containerType,
            modelType, propertyName, attributes);
    }
}

```

默认使用的 `ModelMetadataProvider` 通过 `System.Web.Mvc.ModelMetadataProviders` 来获取。如下面的代码片段所示，`ModelMetadataProviders` 具有一个 `ModelMetadataProvider` 类型的静态可读可写属性 `Current` 用于获取和设置当前使用的 `ModelMetadataProvider`。在默认情况下，`Current` 属性返回的就是一个 `CachedDataAnnotationsModelMetadataProvider` 对象。

```

public class ModelMetadataProviders
{
    public static ModelMetadataProvider Current { get; set; }
}

```

图 4-16 揭示了包括各种 `ModelMetadata` 和对应 `ModelMetadataProvider` 在内的整个 Model 元数据提供系统中各种类型及其相互关系。

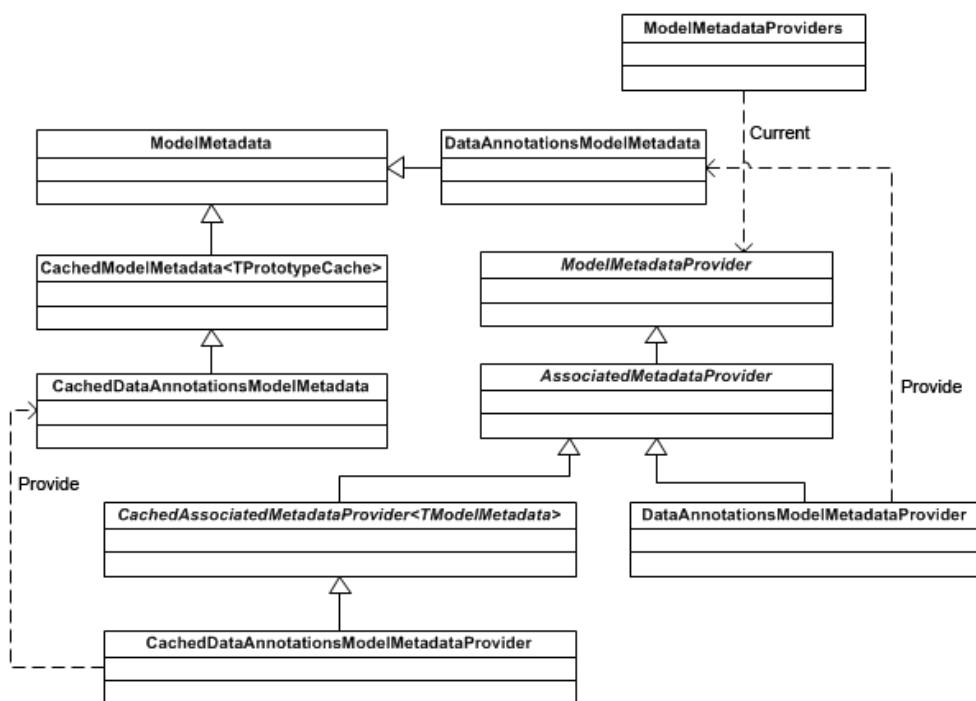


图 4-16 Model 元数据提供系统中的 `ModelMetadata` 和 `ModelMetadataProvider`

ViewData 与 Model 元数据

如下面的代码片段所示，作为 Controller 基类的 ControllerBase 具有一个类型为 System.Web.Mvc.ViewDataDictionary 的 ViewData 对象，用于存储向 View 传递的数据。ViewDataDictionary 具有一个 ModelMetadata 属性返回相应的 Model 元数据，也就是说，不论是在 Controller 还是在 View 中，只要具有一个不为 Null 的 Model 对象（ViewDataDictionary 的 Model 属性），ASP.NET MVC 就可以根据 ViewData 获取针对该 Model 对象的 Model 元数据。

```
public class ViewDataDictionary :
    IDictionary<string, object>,
    ICollection<KeyValuePair<string, object>>,
    IEnumerable<KeyValuePair<string, object>>,
    IEnumerable
{
    //其他成员
    public object Model { get; set; }
    public virtual ModelMetadata ModelMetadata { get; set; }
}

public abstract class ControllerBase : IController
{
    //其他成员
    public ViewDataDictionary ViewData { get; set; }
}
```

4.3.3 Model 元数据提供系统的扩展

对 Model 元数据提供系统的扩展主要体现在对 ModelMetadataProvider 自定义上。基于标注特性的元数据定义方式最终是通过 CachedDataAnnotationsModelMetadataProvider 来实现，通过自定义 ModelMetadataProvider 完全可以提供一种全新的 Model 元数据定义方式。不过经常使用的方式还是让自定义 ModelMetadataProvider 继承 CachedDataAnnotationsModelMetadataProvider 并在现有元数据提供机制上做一些扩展。

实例演示：通过自定义 ModelMetadataProvider 定制 Model 元数据（S413）

在本章的第1节中我们创建了一个用于控制目标元素显示名称的 DisplayTextAttribute 特性（S408）。该特性支持基于资源文件的本地化，并且可以省去对资源条目名称和资源类型的显式指定。该 DisplayTextAttribute 特性是通过实现 IMetadataAware 接口的形式实现的，现在我们将它转换成基于自定义 ModelMetadataProvider 的实现方式。

对于之前定义的 DisplayTextAttribute 特性，只需要对其进行简单的修改。如下面的代码片段所示，我们删除了它实现的 IMetadataAware 接口，将实现的 OnMetadataCreated 方法名改成 SetDisplayName。

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Property)]
public class DisplayTextAttribute : Attribute
```

```

{
    //其他成员
    public void SetDisplayName(ModelMetadata metadata)
    {
        this.DisplayName = this.DisplayName ?? (metadata.PropertyName ??
            metadata.ModelType.Name);
        if (null == this.ResourceType)
        {
            metadata.DisplayName = this.DisplayName;
            return;
        }
        PropertyInfo property = this.ResourceType.GetProperty(this.DisplayName,
            BindingFlags.NonPublic | BindingFlags.Public | BindingFlags.Static);
        metadata.DisplayName = property.GetValue(null, null).ToString();
    }

    public static void SetResourceType(Type resourceType)
    {
        staticResourceType = resourceType;
    }
}

```

为了将 `DisplayTextAttribute` 应用到 Model 元数据的初始化过程中，通过继承 `CachedDataAnnotationsModelMetadataProvider` 创建了如下一个 `ExtendedDataAnnotationsProvider`。在重写的 `CreateMetadataPrototype` 方法中，我们调用基类同名方法创建了作为原型的 `ModelMetadata` 对象，然后根据应用的 `DisplayTextAttribute` 特性对它的 `DisplayName` 属性进行了定制。在重写的 `CreateMetadataFromPrototype` 方法中，我们让最终创建的 `ModelMetadata` 对象与作为原型的 `ModelMetadata` 并具有相同的 `DisplayName` 属性（在默认情况下，总是根据应用的 `DisplayAttribute/DisplayNameAttribute` 特性进行设置）。

```

public class ExtendedDataAnnotationsProvider :
    CachedDataAnnotationsModelMetadataProvider
{
    protected override CachedDataAnnotationsModelMetadata
        CreateMetadataPrototype(IEnumerable<Attribute> attributes,
            Type containerType, Type modelType, string propertyName)
    {
        CachedDataAnnotationsModelMetadata modelMetadata =
            base.CreateMetadataPrototype(attributes, containerType, modelType,
                propertyName);
        if (string.IsNullOrEmpty(modelMetadata.DisplayName))
        {
            DisplayTextAttribute displayTextAttribute = attributes
                .OfType<DisplayTextAttribute>().FirstOrDefault();
            if (null != displayTextAttribute)
            {
                displayTextAttribute.SetDisplayName(modelMetadata);
            }
        }
        return modelMetadata;
    }

    protected override CachedDataAnnotationsModelMetadata
        CreateMetadataFromPrototype(CachedDataAnnotationsModelMetadata prototype,
            Func<object> modelAccessor)
    {
    }
}

```



```

    {
        CachedDataAnnotationsModelMetadata modelMetadata =
            base.CreateMetadataFromPrototype(prototype, modelAccessor);
        modelMetadata.DisplayName = prototype.DisplayName;
        return modelMetadata;
    }
}

```

对于之前创建的演示实例（S408），如果我们在 `Global.asax` 中通过如下的方式对自定义的 `ExtendedDataAnnotationsProvider` 进行注册，该实例应用同样可以正常运行。

```

public class MvcApplication : System.Web.HttpApplication
{
    //其他成员
    protected void Application_Start()
    {
        //其他操作
        DisplayTextAttribute.SetResourceType(typeof(Resources));
        ModelMetadataProviders.Current = new ExtendedDataAnnotationsProvider();
    }
}

```

本章小结

Model 元数据是针对数据类型的一种描述信息，它通过提供针对数据类型本身及其成员属性的描述信息控制数据在界面上的呈现方式。Model 元数据同时也为 Model 绑定和验证提供必不可少的元数据信息。ASP.NET MVC 默认提供基于数据注解特性的 Model 元数据定义方法，除了利用预定义的数据注解特性对目标元素（数据类型或其属性）的 Model 元数据进行定制之外，还可以通过实现 `IMetadataAware` 接口定义相应的特性对最终的 Model 元数据进行灵活控制。

Model 元数据成就了基于模板的数据呈现方式，ASP.NET MVC 提供了一系列默认的模板用于控制目标数据在显示或者编辑模式下 HTML 的生成。还可以通过定义 View 的方式创建自定义的模板，使针对某些类型的数据按照我们希望的方式呈现在界面上。

ASP.NET MVC 具有一个可扩展的以 `ModelMetadataProvider` 为核心的 Model 元数据提供系统，默认采用的 `ModelMetadataProvider` 类型为 `CachedDataAnnotationsModelMetadataProvider`，由它提供的 Model 元数据通过一个 `CachedDataAnnotationsModelMetadata` 表示。除了通过相应的数据注解特性来定制 Model 元数据之外，还可以通过自定义 `ModelMetadataProvider` 的方式来从根本上控制最终提供的 Model 元数据。

第 5 章 Model 的绑定

针对请求的处理和响应最终体现在对激活 Controller 对象中相应 Action 方法的执行上，而 Action 方法执行的前提在于能够根据请求正确地提供相应的参数列表。为目标 Action 方法提供参数列表是 Model 绑定的使命，所以 Model 绑定在整个 ASP.NET MVC 框架体系中具有重要的地位。

5.1 ControllerDescriptor、ActionDescriptor 与 ParameterDescriptor

Model 绑定本质上就是为目标 Action 方法生成参数列表的过程。作为 Action 方法参数的数据存在于当前的 HTTP 请求中，它们可能包含在请求的 URL 中，也可能包含在请求消息的报头或者主体中。究竟哪个部分应该作为 Action 方法某个参数的数据源，这依赖于用于描述参数的元数据信息。Action 方法参数的元数据通过 `System.Web.Mvc.ParameterDescriptor` 来描述，而另外两个相关的类型 `System.Web.Mvc.ControllerDescriptor` 和 `System.Web.Mvc.ActionDescriptor` 则用于描述 Controller 和 Action。

5.1.1 ControllerDescriptor

`ControllerDescriptor` 包含了用于描述某个 Controller 的元数据信息。如下面的代码片段所示，`ControllerDescriptor` 具有三个属性，其中 `ControllerName` 和 `ControllerType` 分别表示 Controller 的名称和类型，前者来源于 URL 路由系统针对请求 URL 的解析得到的路由数据。字符串类型的 `UniqueId` 属性表示 `ControllerDescriptor` 的唯一标识，该标识由 `ControllerDescriptor` 本身的类型、Controller 的类型以及 Controller 的名称三者派生。

```
public abstract class ControllerDescriptor : ICustomAttributeProvider
{
    public virtual object[] GetCustomAttributes(bool inherit);
    public virtual object[] GetCustomAttributes(Type attributeType,
        bool inherit);
    public virtual bool IsDefined(Type attributeType, bool inherit);
    public virtual IEnumerable<FilterAttribute> GetFilterAttributes(
        bool useCache);

    public abstract ActionDescriptor FindAction(
        ControllerContext controllerContext, string actionName);
    public abstract ActionDescriptor[] GetCanonicalActions();

    public virtual string      ControllerName { get; }
    public abstract Type      ControllerType { get; }
    public virtual string      UniqueId { get; }
}

public interface ICustomAttributeProvider
{
    object[] GetCustomAttributes(bool inherit);
    object[] GetCustomAttributes(Type attributeType, bool inherit);
    bool IsDefined(Type attributeType, bool inherit);
}
```

`ControllerDescriptor` 实现了 `System.Reflection.ICustomAttributeProvider` 接口，意味着我们可以通过调用 `GetCustomAttributes` 方法获取应用在 Controller 类型上相应的特性，也可以

调用 `IsDefined` 方法判断指定的自定义特性类型是否应用在对应的 `Controller` 类型上。另一个方法 `GetFilterAttributes` 用于获取应用在 `Controller` 上的所有筛选器特性，我们会在第 7 章“Action 方法的执行”中对筛选器进行详细地介绍。

`ControllerDescriptor` 类型本身并没有通过对 `Controller` 类型实施反射来得到相应的特性。两个 `GetCustomAttributes` 方法均返回一个空的数组对象，`IsDefined` 方法则直接返回 `False`，而方法 `GetFilterAttributes` 返回一个元素类型为 `FilterAttribute` 的空数组对象。

`ControllerDescriptor` 的抽象方法 `FindAction` 根据指定的 `ControllerContext` 得到用于描述指定 Action 的 `ActionDescriptor` 对象，参数 `actionName` 表示 Action 的名称，`GetCanonicalActions` 方法则返回一个 `ActionDescriptor` 数组，代表描述定义在当前 `Controller` 中的所有“候选 Action”。

ReflectedControllerDescriptor

ASP.NET MVC 定义了一个 `System.Web.Mvc.ReflectedControllerDescriptor` 类型，它是抽象类型 `ControllerDescriptor` 的唯一继承者。`ReflectedControllerDescriptor` 通过对 `Controller` 类型的反射得到用于描述 `Controller` 的元数据。如下面的代码片段所示，表示 `Controller` 类型的 `ControllerType` 属性在构造函数中指定。

```
public class ReflectedControllerDescriptor : ControllerDescriptor
{
    public ReflectedControllerDescriptor(Type controllerType);

    public override object[] GetCustomAttributes(bool inherit);
    public override object[] GetCustomAttributes(Type attributeType,
        bool inherit);
    public override bool IsDefined(Type attributeType, bool inherit);
    public override IEnumerable<FilterAttribute> GetFilterAttributes(
        bool useCache);

    public override ActionDescriptor FindAction(
        ControllerContext controllerContext, string actionName);
    public override ActionDescriptor[] GetCanonicalActions();

    public sealed override Type ControllerType { get; }
}
```

由于基类 `ControllerDescriptor` 并没有真正实现定义在接口 `ICustomAttributeProvider` 中的三个方法，用于获取筛选器特性列表的 `GetFilterAttributes` 方法返回的也是一个空的 `FilterAttribute` 数组，所以 `ReflectedControllerDescriptor` 重写了这些方法。

`GetCanonicalActions` 方法返回的 `ActionDescriptor` 数组通过对定义在 `Controller` 类型的 Action 方法进行反射得到。对于定义在当前 `Controller` 类型中的所有方法成员来说，能够被视为 Action 方法的仅限于“公有”的“实例”方法，但并不包括从抽象类 `Controller` 中继承下来的方法。

在默认情况下，方法的名称被视为 Action 的名称，所以当调用 `FindAction` 方法的时候，

ASP.NET MVC 会从描述“候选 Action”的 `ActionDescriptor` 列表中筛选出方法名称相匹配(不区分大小写)的 `ActionDescriptor` 对象。但是我们并不一定需要将两者 (Action 名称和方法名称) 强行绑定在一起, 为了让 Action 名称能够独立于目标方法名称, 可以在 Action 方法上应用具有如下定义的 `System.Web.Mvc.ActionNameAttribute` 特性来指定 Action 的名称。

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple=false, Inherited=true)]
public sealed class ActionNameAttribute : ActionNameSelectorAttribute
{
    public ActionNameAttribute(string name);
    public override bool IsValidName(ControllerContext controllerContext,
        string actionName, MethodInfo methodInfo);
    public string Name { get; }
}
```

`ActionNameAttribute` 继承自具有如下定义的 `System.Web.Mvc.ActionNameSelectorAttribute` 抽象类。`ActionNameSelectorAttribute` 通过其抽象方法 `IsValidName` 判断指定的 Action 名称是否与目标 Action 方法相匹配。定义在 `ActionNameAttribute` 中的 `IsValidName` 方法的逻辑很简单, 它仅仅判断指定的 Action 名称是否和指定的名称一致 (忽略大小写)。

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false,
    Inherited = true)]
public abstract class ActionNameSelectorAttribute : Attribute
{
    public abstract bool IsValidName(ControllerContext controllerContext,
        string actionName, MethodInfo methodInfo);
}
```

很多人会将 `ActionNameSelectorAttribute` 与 `System.Web.Mvc.ActionMethodSelectorAttribute` (定义如下) 混淆, 虽然两者都具有对候选 Action 进行筛选的作用, 不过前者主要针对 Action 名称进行筛选, 后者通过抽象的方法 `IsValidForRequest` 判断目标 Action 方法是否与当前的请求相匹配。

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false,
    Inherited = true)]
public abstract class ActionMethodSelectorAttribute : Attribute
{
    public abstract bool IsValidForRequest(ControllerContext controllerContext,
        MethodInfo methodInfo);
}
```

ASP.NET MVC 定义了如下 7 个基于相应 HTTP 方法 (GET、POST、PUT、DELETE、Head、Options 和 Patch) 的 `ActionMethodSelectorAttribute` 类型。当将它们应用到某个 Action 方法上时, 只有在当前请求的 HTTP 方法与之相匹配的情况下目标 Action 方法才会被选择。

- `System.Web.Mvc.HttpGetAttribute`
- `System.Web.Mvc.HttpPostAttribute`
- `System.Web.Mvc.HttpPutAttribute`

- System.Web.Mvc.HttpDeleteAttribute
- System.Web.Mvc.HttpHeadAttribute
- System.Web.Mvc.HttpOptionsAttribute
- System.Web.Mvc.HttpPatchAttribute

除了上面 7 个基于某种 HTTP 方法的 `ActionMethodSelectorAttribute` 特性之外, ASP.NET MVC 还定义了另一个名为 `System.Web.Mvc.AcceptVerbsAttribute` 的特性。`AcceptVerbsAttribute` 的不同之处在于可以指定多个匹配的 HTTP 方法。如下面的代码片段所示, `AcceptVerbsAttribute` 具有一个 `ICollection<string>` 类型的只读属性 `Verbs`, 表示目标 Action 方法支持的 HTTP 方法列表 (HTTP Method 又被称为 HTTP Verb), 该属性在构造函数中被初始化。实际上述的 7 个 `ActionMethodSelectorAttribute` 在内部使用了 `AcceptVerbsAttribute` 特性实现了具体的 Action 方法选择逻辑。

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple=false, Inherited=true)]
public sealed class AcceptVerbsAttribute : ActionMethodSelectorAttribute
{
    public AcceptVerbsAttribute(HttpVerbs verbs);
    public AcceptVerbsAttribute(params string[] verbs);

    public override bool IsValidForRequest(ControllerContext controllerContext,
        MethodInfo methodInfo);
    public ICollection<string> Verbs { get; }
}

[Flags]
public enum HttpVerbs
{
    Get      = 1,
    Post     = 2,
    Put      = 4,
    Delete   = 8,
    Head     = 16,
}
```

从上面的代码片段可以看出, `AcceptVerbsAttribute` 具有两个构造函数, 其参数类型分别是 `System.Web.Mvc.HttpVerbs` 枚举和字符串数组。由于 `AcceptVerbsAttribute` 枚举应用了 `FlagsAttribute` 特性, 可以使用操作符 “|” 指定多个 HTTP 方法。如下所示的两种在 Action 方法 `UpdateContact` 上应用 `AcceptVerbsAttribute` 特性的方式是等效的。顺便提一下, 通过字符串指定的 HTTP 方法是不区分大小写的。

```
//使用 HttpVerbs 枚举表示 HTTP 方法
public class ContactController
{
    [AcceptVerbs(HttpVerbs.Put|HttpVerbs.Post|HttpVerbs.Delete)]
    public ActionResult UpdateContact(Contact contact)
    {
        //省略实现
    }
}
```

```
//使用字符串表示 HTTP 方法
public class ContactController
{
    [AcceptVerbs("PUT", "POST", "DELETE")]
    public ActionResult UpdateContact(Contact contact)
    {
        //省略实现
    }
}
```

除了上面 8 个基于 HTTP 方法的 `ActionMethodSelectorAttribute` 特性, ASP.NET MVC 还定义了另一个具有如下定义的 `System.Web.Mvc.NonActionAttribute` 特性。顾名思义, 应用了 `NonActionAttribute` 特性的方法将不会被认为是一个 Action 方法。由于其 `IsValidForRequest` 方法直接返回 `False`, 所以在根据请求进行目标 Action 方法选择的时候, 这样的方法总是被排除在候选范围之内。

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false,
    Inherited = true)]
public sealed class NonActionAttribute : ActionMethodSelectorAttribute
{
    public override bool IsValidForRequest(ControllerContext controllerContext,
        MethodInfo methodInfo)
    {
        return false;
    }
}
```

当 `FindAction` 方法执行的时候, 候选 `ActionDescriptor` 列表中与请求 Action 名称相匹配的 `ActionDescriptor` 被筛选出来, 然后利用应用在对应方法上的 `ActionNameSelectorAttribute` 和 `ActionMethodSelectorAttribute` 特性做进一步筛选。如果最终没有一个符合条件的 `ActionDescriptor`, 该方法会返回 `Null` (最终会抛出一个状态码为 404 的 `HttpException` 异常)。如果具有多个匹配的 `ActionDescriptor`, 该方法会直接抛出 `System.Reflection.AmbiguousMatchException` 异常, 也就是说对于每一次请求, 要求有且只有一个匹配的 `ActionDescriptor`。

ReflectedAsyncControllerDescriptor

`System.Web.Mvc.Async.ReflectedAsyncControllerDescriptor` 类型为 `ReflectedControllerDescriptor` 的异步版本, 如下面的代码片段所示, 两者具有类似的成员定义。实际上除了 `FindAction` 和 `GetCanonicalActions` 两个方法外, 其他方法的实现逻辑与 `ReflectedControllerDescriptor` 完全一致。

```
public class ReflectedAsyncControllerDescriptor : ControllerDescriptor
{
    public ReflectedAsyncControllerDescriptor(Type controllerType);

    public override object[] GetCustomAttributes(bool inherit);
    public override object[] GetCustomAttributes(Type attributeType,
        bool inherit);
    public override IEnumerable<FilterAttribute> GetFilterAttributes()
```

```

        bool useCache);
    public override bool IsDefined(Type attributeType, bool inherit);

    public override ActionDescriptor FindAction(
        ControllerContext controllerContext, string actionName);
    public override ActionDescriptor[] GetCanonicalActions();

    public sealed override Type ControllerType { get; }
}

```

ReflectedAsyncControllerDescriptor 的 GetCanonicalActions 总是返回一个空的 ActionDescriptor 数组。对于继承自 AsyncController 的 Controller 类型，一个异步 Action 方法由两个匹配的方法 XxxAsync/XxxCompleted 构成，FindAction 方法在根据方法名称进行匹配的时候会自动忽略掉方法名称的“Async”和“Completed”后缀。

5.1.2 ActionDescriptor

用于描述 Action 的 ActionDescriptor 对象对应着定义在 Controller 类型中的某个 Action 方法。如下面的代码片段所示，它具有 ActionName 和 ControllerDescriptor 两个抽象只读属性，它们分别表示 Action 名称和描述所在 Controller 的 ControllerDescriptor 对象。表示唯一标识的 UniqueId 属性由自身类型、Controller 的类型与 Action 名称三者派生。

```

public abstract class ActionDescriptor : ICustomAttributeProvider
{
    public virtual object[] GetCustomAttributes(bool inherit);
    public virtual object[] GetCustomAttributes(Type attributeType,
        bool inherit);
    public virtual bool IsDefined(Type attributeType, bool inherit);
    public virtual IEnumerable<FilterAttribute> GetFilterAttributes(
        bool useCache);

    public abstract ParameterDescriptor[] GetParameters();
    public abstract object Execute(ControllerContext controllerContext,
        IDictionary<string, object> parameters);
    public virtual ICollection<ActionSelector> GetSelectors();
    public virtual FilterInfo GetFilters();

    public abstract string ActionName { get; }
    public abstract ControllerDescriptor ControllerDescriptor { get; }
    public virtual string UniqueId { get; }
}

```

ActionDescriptor 同样实现了 ICustomAttributeProvider 接口，可以通过方法 GetCustomAttributes 得到应用在 Action 方法上的相关特性，或者借助 IsDefined 方法判断某种指定的特性类型是否应用在对应的 Action 方法上。GetFilterAttributes 方法用于返回应用在 Action 方法上的所有筛选器特性。与 ControllerDescriptor 一样，ActionDescriptor 并没有真正采用反射去解析应用在 Action 方法上的特性，它的 GetCustomAttributes 和 GetFilterAttributes 方法返回的都是一个空数组，而 IsDefined 方法直接返回 False。

Action 方法的每一个参数通过一个 `ParameterDescriptor` 对象来描述, `ActionDescriptor` 的抽象方法 `GetParameters` 返回一个描述所有参数的 `ParameterDescriptor` 数组。另一个重要的抽象 `Execute` 方法实现了最终对 Action 的执行, 该方法两个参数分别表示当前 `ControllerContext` 和参数列表。

`GetSelectors` 返回一组 `ActionSelector` 对象, 实际上是一组 `System.Web.Mvc.ActionSelector` 类型的委托对象。如下面的代码片段所示, `ActionSelector` 委托具有一个类型为 `ControllerContext` 的参数, 布尔类型的返回值表示目标 Action 方法是否与指定的 `ControllerContext` 相匹配。该方法默认返回的是一个空的 `ActionSelector` 集合。

```
public delegate bool ActionSelector(ControllerContext controllerContext);
```

`ActionDescriptor` 的 `GetFilters` 方法返回的是一个 `System.Web.Mvc.FilterInfo` 对象, 可以通过它得到应用在该 Action 方法上所有的筛选器。如下面的代码所示, `FilterInfo` 具有四个只读的集合属性, 分别表示四种类型的筛选器 (`ActionFilter`、`AuthorizationFilter`、`ExceptionFilter` 和 `ResultFilter`)。在这里返回的是一个空的 `FilterInfo` 对象, 它的四个集合属性并不包含任何元素。

```
public class FilterInfo
{
    public IList<IActionFilter> ActionFilters { get; }
    public IList<IAuthorizationFilter> AuthorizationFilters { get; }
    public IList<IExceptionFilter> ExceptionFilters { get; }
    public IList<IResultFilter> ResultFilters { get; }
}
```

AsyncActionDescriptor

异步版本的 `ActionDescriptor` 通过 `System.Web.Mvc.Async.AsyncActionDescriptor` 类型表示, 如下面的代码片段所示, `AsyncActionDescriptor` 继承自抽象类 `ActionDescriptor`。除了重写 `Execute` 方法之外, 它还定义了两个用于异步执行 Action 的抽象方法 `BeginExecute/EndExecute`。

```
public abstract class AsyncActionDescriptor : ActionDescriptor
{
    public abstract IAsyncResult BeginExecute(
        ControllerContext controllerContext,
        IDictionary<string, object> parameters, AsyncCallback callback,
        object state);
    public abstract object EndExecute(IAsyncResult asyncResult);

    public override object Execute(ControllerContext controllerContext,
        IDictionary<string, object> parameters);
}
```

实际上 `AsyncActionDescriptor` 重写的 `Execute` 方法并没有真正去指定 Action, 而是直接抛出一个 `InvalidOperationException` 异常, 意味着异步方法不能通过 `Execute` 方法以同步方式执行。

ReflectedActionDescriptor

ReflectedControllerDescriptor 的 FindAction 和 GetCanonicalActions 方法返回的 ActionDescriptor 对象类型为 System.Web.Mvc.ReflectedActionDescriptor 对象，它针对 Action 元数据信息的解析同样也是通过对目标方法进行反射实现的。如下面的代码片段所示，ReflectedActionDescriptor 直接继承自 ActionDescriptor，分别表示 Action 名称、所在 Controller 的描述以及 Action 方法的只读属性 ActionName、ControllerDescriptor 和 MethodInfo 均在构造函数中初始化。

```
public class ReflectedActionDescriptor : ActionDescriptor
{
    public ReflectedActionDescriptor(MethodInfo methodInfo,
        string actionName, ControllerDescriptor controllerDescriptor);

    public override object[] GetCustomAttributes(bool inherit);
    public override object[] GetCustomAttributes(Type attributeType,
        bool inherit);
    public override bool IsDefined(Type attributeType, bool inherit);
    public override IEnumerable<FilterAttribute> GetFilterAttributes(
        bool useCache);

    public override ParameterDescriptor[] GetParameters();
    public override object Execute(ControllerContext controllerContext,
        IDictionary<string, object> parameters);
    public override ICollection<ActionSelector> GetSelectors();

    public override string ActionName { get; }
    public override ControllerDescriptor ControllerDescriptor { get; }
    public override MethodInfo MethodInfo { get; }
    public override string UniqueId { get; }
}
```

由于基类 ActionDescriptor 并没有真正地实现定义在接口 ICustomAttributeProvider 中的三个方法，获取筛选器特性列表的 GetFilters 方法返回的也只是一个空的 FilterInfo 对象，所以 ReflectedActionDescriptor 重写了这些方法。

ReflectedActionDescriptor 还重写了 UniqueId 属性，在现有的基础上将表示 Action 方法的 MethodInfo 对象作为了决定元素之一。也就是说，作为唯一标识的 UniqueId 属性通过自身的类型、Controller 类型、Action 名称和表示目标 Action 方法的 MethodInfo 对象四者派生。

对于通过方法 GetParameters 返回的用于描述所有参数的 ParameterDescriptor 数组，也是通过对 Action 方法的参数列表进行反射来创建的。Execute 方法最终传入参数列表调用 MethodInfo 对象执行 Action 方法。它的 GetSelectors 方法返回一组应用在 Action 方法上的 ActionMethodSelectorAttribute 特性列表。

ReflectedAsyncActionDescriptor

异步的 ReflectedControllerDescriptor 由 System.Web.Mvc.Async.ReflectedAsyncAction

Descriptor 类型表示，它用于描述定义在 AsyncController 中以 XxxAsync/XxxCompleted 形式定义的异步 Action。一个 ReflectedAsyncActionDescriptor 对象通过代表这两个方法的 MethodInfo 对象来创建。如下面的代码片段所示，ReflectedAsyncActionDescriptor 的构造的参数 asyncMethodInfo 和 completedMethodInfo 就代表这两个 MethodInfo。在构造函数中初始化的这两个 MethodInfo 对象分别通过只读属性 AsyncMethodInfo 和 CompletedMethodInfo 返回。

```
public class ReflectedAsyncActionDescriptor : AsyncActionDescriptor
{
    public ReflectedAsyncActionDescriptor(MethodInfo asyncMethodInfo,
        MethodInfo completedMethodInfo, string actionName,
        ControllerDescriptor controllerDescriptor);

    public override IAsyncResult BeginExecute(
        ControllerContext controllerContext,
        IDictionary<string, object> parameters, AsyncCallback callback,
        object state);
    public override object EndExecute(IAsyncResult asyncResult);

    public override object[] GetCustomAttributes(bool inherit);
    public override object[] GetCustomAttributes(Type attributeType,
        bool inherit);
    public override IEnumerable<FilterAttribute> GetFilterAttributes(
        bool useCache);
    public override ParameterDescriptor[] GetParameters();
    public override ICollection<ActionSelector> GetSelectors();
    public override bool IsDefined(Type attributeType, bool inherit);

    public override string                ActionName { get; }
    public MethodInfo                     AsyncMethodInfo { get; }
    public MethodInfo                     CompletedMethodInfo { get; }
    public override ControllerDescriptor ControllerDescriptor { get; }
    public override string                 UniqueId { get; }
}
```

ReflectedAsyncActionDescriptor 用于特性解析（定义在 ICustomAttributeProvider 接口中的三个方法、用于获取筛选器特性列表的 GetFilterAttributes 方法以及获取 ActionMethod SelectorAttribute 特性列表的 GetSelectors 方法）和参数描述解析（GetParameters 方法）的方法都是通过对 XxxAsync 方法（对应于 AsyncMethodInfo 属性）的反射实现的。它的 BeginExecute/EndExecute 方法最终利用了 AsyncMethodInfo 和 CompletedMethodInfo 实现了对 Action 的异步执行。

TaskAsyncActionDescriptor

异步 Action 除了以配对的 XxxAsync/XxxCompleted 方法进行定义之外，还可以通过一个返回类型为 System.Threading.Tasks.Task 的方法来定义。前者必须定义在 AsyncController 中，后者则可以定义在普通的 Controller 中并通过 System.Web.Mvc.Async.TaskAsync ActionDescriptor 对象来描述。如下面的代码片段所示，TaskAsyncActionDescriptor 通过一个类型为 MethodInfo 的只读属性 TaskMethodInfo 表示异步 Action 方法，该属性在构造函数中初始化。

```

public class TaskAsyncActionDescriptor : AsyncActionDescriptor
{
    public TaskAsyncActionDescriptor(MethodInfo taskMethodInfo,
        string actionName, ControllerDescriptor controllerDescriptor);

    public override IAsyncResult BeginExecute(
        ControllerContext controllerContext,
        IDictionary<string, object> parameters, AsyncCallback callback,
        object state);
    public override object EndExecute(IAsyncResult asyncResult);
    public override object Execute(ControllerContext controllerContext,
        IDictionary<string, object> parameters);

    public override object[] GetCustomAttributes(bool inherit);
    public override object[] GetCustomAttributes(Type attributeType,
        bool inherit);
    public override IEnumerable<FilterAttribute> GetFilterAttributes(
        bool useCache);
    public override ParameterDescriptor[] GetParameters();
    public override ICollection<ActionSelector> GetSelectors();
    public override bool IsDefined(Type attributeType, bool inherit);

    public override string ActionName { get; }
    public override ControllerDescriptor ControllerDescriptor { get; }
    public override MethodInfo TaskMethodInfo { get; }
    public override string UniqueId { get; }
}

```

`TaskAsyncActionDescriptor` 对特性解析和参数描述解析的方法都是通过针对 `TaskMethodInfo` 的反射来完成的，用于异步执行 Action 操作的 `BeginExecute/EndExecute` 方法也是借助于这个 `MethodInfo` 对象实现的。`TaskAsyncActionDescriptor` 重写了 `Execute` 方法并在其中直接抛出异常。

不论是 Controller 还是 Action，都具有同步和异步两个版本，至于异步 Controller 和两种形式的异步 Action 方法如何定义，以及同步和异步 `ControllerDescriptor/ActionDescriptor` 分别在什么情况下被创建，我们将在第 7 章“Action 的执行”中进行详细介绍。

5.1.3 ParameterDescriptor

Model 绑定可以看成是为目标 Action 的方法生成参数列表的过程，所以针对参数的元数据描述才是 Model 绑定的核心依据。服务于 Model 绑定的参数元数据通过 `ParameterDescriptor` 类型来表示，而 `ActionDescriptor` 的 `GetParameters` 方法返回的就是一个 `ParameterDescriptor` 数组。

如下面的代码片段所示，`ParameterDescriptor` 同样实现了 `ICustomAttributeProvider` 接口以提供应用在参数上的特性。与抽象类型 `ControllerDescriptor` 和 `ActionDescriptor` 一样，针对参数的特性解析并没有真正实现在 `ParameterDescriptor` 相应的方法中，所以 `GetCustomAttributes` 方法返回的依旧是一个空数组，而 `False` 还是直接作为 `IsDefined` 方法的返回值。

```

public abstract class ParameterDescriptor : ICustomAttributeProvider
{
    public virtual object[] GetCustomAttributes(bool inherit);
    public virtual object[] GetCustomAttributes(Type attributeType,
        bool inherit);
    public virtual bool IsDefined(Type attributeType, bool inherit);

    public abstract ActionDescriptor ActionDescriptor { get; }
    public abstract string ParameterName { get; }
    public abstract Type ParameterType { get; }
    public virtual object DefaultValue { get; }

    public virtual ParameterBindingInfo BindingInfo { get; }
}

```

`ParameterDescriptor` 的只读属性 `ActionDescriptor` 表示描述所在 `Action` 的 `ActionDescriptor` 对象。属性 `ParameterName`、`ParameterType` 和 `DefaultValue` 分别表示参数的名称、类型和默认值。`ParameterDescriptor` 的只读属性 `BindingInfo` 表示的 `System.Web.Mvc.ParameterBindingInfo` 对象用于控制请求数据与参数的绑定行为。如下面的代码片段所示，抽象类 `ParameterBindingInfo` 具有四个属性，其中类型为 `IModelBinder` 的 `Binder` 属性返回的 `ModelBinder` 对象是整个 `Model` 绑定的核心，我们将在本章后续部分进行单独介绍。

```

public abstract class ParameterBindingInfo
{
    public virtual IModelBinder Binder { get; }

    public virtual ICollection<string> Include { get; }
    public virtual ICollection<string> Exclude { get; }
    public virtual string Prefix { get; }
}

```

如果参数类型是一个复杂类型，默认情况下会绑定其所有公共可写属性，而两个 `ICollection<string>` 类型的属性 `Include` 和 `Exclude` 表示显式设置的参与/不参与绑定的属性名称列表。在默认情况下，请求数据与参数之间严格按照名称进行绑定，但是有时候请求数据名称具有相应的前缀，这个前缀体现在 `ParameterBindingInfo` 的 `Prefix` 属性上。

ReflectedParameterDescriptor

默认使用的 `ParameterBindingInfo` 是通过对目标参数对应的 `ParameterInfo` 对象进行反射获得的，这样的 `ParameterDescriptor` 通过具有如下定义的 `System.Web.Mvc.ReflectedParameterDescriptor` 类型表示，而这个 `ParameterInfo` 对象通过只读属性 `ParameterInfo` 表示。

```

public class ReflectedParameterDescriptor : ParameterDescriptor
{
    public ReflectedParameterDescriptor(ParameterInfo parameterInfo,
        ActionDescriptor actionDescriptor);
    public override object[] GetCustomAttributes(bool inherit);
    public override object[] GetCustomAttributes(Type attributeType,
        bool inherit);
    public override bool IsDefined(Type attributeType, bool inherit);

    public override ActionDescriptor ActionDescriptor { get; }
}

```

```

public override ParameterBindingInfo BindingInfo { get; }
public override object DefaultValue { get; }
public override string ParameterName { get; }
public override Type ParameterType { get; }

public ParameterInfo ParameterInfo { get; }
}

```

ReflectedParameterDescriptor 的 BindingInfo 属性返回的是一个 ReflectedParameterBindingInfo 对象，这是一个内部类型。该 BindingInfo 对象的 Include、Exclude 和 Prefix 属性来源于应用在参数上的 System.Web.Mvc.BindAttribute 特性。如下面的代码片段所示，BindAttribute 中同样定义了这三个属性，其中 Include 和 Exclude 为通过逗号作为分隔符的属性名称列表。具有布尔返回类型的 IsPropertyAllowed 方法用于判断指定的属性是否允许绑定，只有当指定的属性名在 Include 列表中（或者 Include 列表为空）并且不在 Exclude 列表中的情况下，该属性才返回 True。

```

[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Class,
    AllowMultiple=false, Inherited=true)]
public sealed class BindAttribute : Attribute
{
    public bool IsPropertyAllowed(string propertyName);

    public string Include { get; set; }
    public string Exclude { get; set; }
    public string Prefix { get; set; }
}

```

在本节中，我们介绍了三个重要的描述类型，即分别描述 Controller、Action 和参数的 ControllerDescriptor、ActionDescriptor 和 ParameterDescriptor。旨在生成参数的 Model 绑定主要使用 ParameterDescriptor，但是其他两个 ControllerDescriptor 和 ActionDescriptor 在整个 ASP.NET MVC 框架体系中同样具有重要的作用。如图 5-1 所示的 UML 体现了所有抽象和具体描述类型之间的关系。

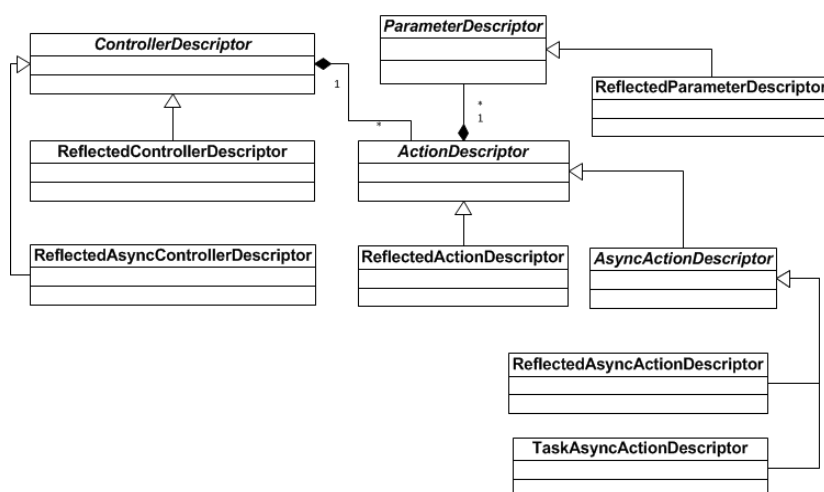


图 5-1 ControllerDescriptor – ActionDescriptor – ParameterDescriptor

5.2 ValueProvider

Model 绑定的数据具有多个来源，可能来源于提交的表单或者 JSON 字符串，也可能来源于当前的路由数据，或者来源于请求地址的查询字符串。ASP.NET MVC 将这种基于不同数据来源的数据提供机制实现在一个叫做 ValueProvider 的组件中。

一般来讲，一个 ValueProvider 采用的数据源容器具有类似于字典类型的数据结构，通过指定相应的 Key 从字典中获取相应的数据。ASP.NET MVC 下的 ValueProvider 实现了具有如下定义的接口 System.Web.Mvc.IValueProvider。GetValue 方法根据指定的 Key 从数据源中获取对应的数据，但是这个 Key 与存在于数据源容器中对数据条目的 Key 可能并非完全一致，后者可能在前者基础上添加相应的前缀，而 ContainsPrefix 方法正是用于判断数据源容器中是否具有包含指定前缀的 Key。Controller 使用的 ValueProvider 可以通过定义在 ControllerBase 中的 ValueProvider 属性进行获取和设置。

```
public interface IValueProvider
{
    bool ContainsPrefix(string prefix);
    ValueProviderResult GetValue(string key);
}

public abstract class ControllerBase : IController
{
    //其他成员
    public IValueProvider ValueProvider { get; set; }
}
```

IValueProvider 的 GetValue 方法返回的是一个 System.Web.Mvc.ValueProviderResult 对象，提供的数据包含在该对象之中。如下面的代码片段所示，ValueProviderResult 具有三个只读属性，其中 RawValue 表示提供的原始数据，而 AttemptedValue 表示数据值的字符串表示。

```
[Serializable]
public class ValueProviderResult
{
    public ValueProviderResult(object rawValue, string attemptedValue,
        CultureInfo culture);

    public object ConvertTo(Type type);
    public virtual object ConvertTo(Type type, CultureInfo culture);

    public string      AttemptedValue { get; }
    public CultureInfo Culture { get; }
    public object      RawValue { get; }
}
```

ValueProviderResult 提供了两个 ConvertTo 方法重载以实现向指定目标类型的转换。某些类型（比如时间、日期和货币等）的格式化依赖于相应的语言文化，这个辅助格式化的语言文化通过 culture 参数来指定。如果调用 ConvertTo 方法时没有显式地指定这个参数，ASP.NET MVC 会默认使用 ValueProviderResult 的 Culture 属性值表示的 CultureInfo 对象。

5.2.1 NameValueCollectionValueProvider

ValueProvider 的数据容器一般具有类似于字典的结构。NameValueCollection 表示一种 Key 和 Value 均为字符串的字典，并且对 Key 不具有唯一性约束（即两个元素可以共享相同的 Key）。具有如下定义的 System.Web.Mvc.NameValueCollectionValueProvider 是将一个 NameValueCollection 对象作为数据源容器的 ValueProvider，作为数据源容器的 NameValueCollection 对象将在构造函数中指定。

```
public class NameValueCollectionValueProvider : IUnvalidatedValueProvider,
    IEnumerableValueProvider, IValueProvider
{
    //其他成员
    public NameValueCollectionValueProvider(NameValueCollection collection,
        CultureInfo culture);

    public virtual bool ContainsPrefix(string prefix);
    public virtual IDictionary<string, string> GetKeysFromPrefix(string prefix);
    public virtual ValueProviderResult GetValue(string key);
    public virtual ValueProviderResult GetValue(string key, bool skipValidation);
}

public interface IEnumerableValueProvider : IValueProvider
{
    IDictionary<string, string> GetKeysFromPrefix(string prefix);
}

public interface IUnvalidatedValueProvider : IValueProvider
{
    ValueProviderResult GetValue(string key, bool skipValidation);
}
```

NameValueCollectionValueProvider 除了实现 IValueProvider 接口之外，还实现了两个额外的接口——IEnumerableValueProvider 和 IUnvalidatedValueProvider。从接口命名也可以猜出 IEnumerableValueProvider 主要用于针对目标类型为集合的数据提供，方法 GetKeysFromPrefix 以字典的形式返回数据源容器中所有具有指定前缀的 Key。默认的情况下数据提供会进行数据验证，而 IUnvalidatedValueProvider 接口提供了一个额外的 GetValue 方法使我们可以忽略对数据的验证。

两种前缀形式

辅助实现 Model 绑定的数据提供机制借助了 Model 元数据对数据类型的描述，通过第 4 章“Model 元数据的提供”我们知道描述一个复杂数据类型的 ModelMetadata 具有树型层次化结构，但是作为数据源容器的 NameValueCollection 对象却是一个“扁平”的结构，两者之间的匹配通过前缀来实现。

举个简单的例子，假设通过 NameValueCollectionValueProvider 提供对象的目标类型为具有如下定义的 Contact。表示联系地址的属性是一个复杂类型 Address，那么针对 Contact 类

型的 Model 元数据树具有两个层级。

```
public class Contact
{
    public string    Name { get; set; }
    public string    PhoneNo { get; set; }
    public string    EmailAddress { get; set; }
    public Address   Address { get; set; }
}

public class Address
{
    public string Province { get; set; }
    public string City { get; set; }
    public string District { get; set; }
    public string Street { get; set; }
}
```

由于组成 `NameValueCollection` 的元素值都是字符串，它不可能单独表示一个复杂对象，一个复杂对象需要通过多个元素值组装而成。如果通过 `NameValueCollectionValueProvider` 来初始化一个完整的 `Contact` 对象，表示数据源的 `NameValueCollection` 至少需要包含 7 个元素，分别对应 `Contact` 对象的 `Name`、`PhoneNo` 和 `EmailAddress` 属性，以及 `Address` 对象的 `Province`、`City`、`District` 和 `Street` 属性。两类元素在 `NameValueCollection` 中通过基于属性名称的前缀来区分，具体的结构如表 5-1 所示。

表 5-1 复杂类型数据在 `NameValueCollection` 中的表示

Key	Value
Name	张三
PhoneNo	123456789
EmailAddress	zhangsan@gmail.com
Address.Province	江苏
Address.City	苏州
Address.District	工业园区
Address.Street	星湖街 328 号

将点号(.)作为分隔符的前缀除了表示基于属性的层级关系之外，还可以用于数据筛选。如下面的代码片段所示，我们在 `ContactController` 中定义了一个用于添加联系人的 `Action` 方法 `AddContacts`，它具有两个 `Contact` 类型的参数 `foo` 和 `bar`，表示添加的两个不同的联系人。

```
public class ContactController
{
    public void AddContacts(Contact foo, Contact bar)
    {
        //省略实现
    }
}
```

如果我们采用 `NameValueCollectionValueProvider` 来提供作为 `AddContacts` 方法参数的两个 `Contact` 对象，保存在 `NameValueCollection` 的数据元素必须能够与它们进行合理映射。这样的映射可以通过针对参数名的前缀来实现，具体数据结构如表 5-2 所示。

表 5-2 具有不同前缀的相同类型数据在 `NameValueCollection` 中的表示

Key	Value
foo.Name	张三
foo.PhoneNo	123456789
foo.EmailAddress	zhangsan@gmail.com
foo.Address.Province	江苏
foo.Address.City	苏州
foo.Address.District	工业园区
foo.Address.Street	星湖街 328 号
bar.Name	李四
bar.PhoneNo	987654321
bar.EmailAddress	lisi@gmail.com
bar.Address.Province	江苏
bar.Address.City	苏州
bar.Address.District	工业园区
bar.Address.Street	机场路 328 号

除了采用基于 “.” 的前缀之外，目标类型为数组或集合的数据源元素可以采用基于 “索引” 的前缀，这样的前缀通过方括号 “[]” 表示。表 5-3 体现了表示目标类型为 `Contact` 数组或者集合的数据源容器的结构。

表 5-3 数组/集合对象在 `NameValueCollection` 中的表示（1）

Key	Value
[0].Name	张三
[0].PhoneNo	123456789
[0].EmailAddress	zhangsan@gmail.com
[1].Name	李四
[1].PhoneNo	987654321
[1].EmailAddress	lisi@gmail.com
...	...

除了采用数字作为索引之前，还可以按照如表 5-4 所示的方式通过文字作为索引。两种不同形式的索引对应着不同的 `Model` 绑定机制，我们会在本章后续的部分对两者之间的差异进行详细讲述。

表 5-4 数组/集合对象在 NameValueCollection 中的表示 (2)

Key	Value
[foo].Name	张三
[foo].PhoneNo	123456789
[foo].EmailAddress	zhangsan@gmail.com
[bar].Name	李四
[bar].PhoneNo	987654321
[bar].EmailAddress	lisi@gmail.com
...	...

实例演示：返回指定前缀的 Key (S501、S502)

在了解了这两种不同类型的前缀之后，我们来关注一下 NameValueCollectionValueProvider 实现的 GetKeysFromPrefix 方法。该方法返回的是一个 IDictionary<string, string> 对象，针对一个作为前缀的字符串，该方法返回的字典会包含怎样的数据元素呢？不妨通过一个简单的演示实例来找到答案。

在一个 ASP.NET MVC 应用中定义了如下一个 HomeController。在 Action 方法 Index 中创建了一个 NameValueCollection 对象，并在其中添加了 7 个元素。通过前面对 Contact 的定义我们知道这 7 个元素可以最终组装成一个完整的 Contact 对象，这些元素均以字符串“foo”为前缀。最后针对这个 NameValueCollection 创建了对应的 NameValueCollectionValueProvider，并将其作为 Model 在默认 View 中呈现出来。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        NameValueCollection datasource = new NameValueCollection();

        datasource.Add("foo.Name", "Foo");
        datasource.Add("foo.PhoneNo", "123456789");
        datasource.Add("foo.EmailAddress", "Foo@gmail.com");

        datasource.Add("foo.Address.Province", "江苏");
        datasource.Add("foo.Address.City", "苏州");
        datasource.Add("foo.Address.District", "工业园区");
        datasource.Add("foo.Address.Street", "星湖街 328 号");

        NameValueCollectionValueProvider valueProvider =
            new NameValueCollectionValueProvider(datasource,
            CultureInfo.InvariantCulture);
        return View(valueProvider);
    }
}
```

如下所示是 Action 方法中 Index 对应 View 的定义，这是一个 Model 类型为 NameValueCollectionValueProvider 的强类型 View。在该 View 中，我们分别将“foo”和“foo.Address”作为前缀调用 NameValueCollectionValueProvider 对象 GetKeysFromPrefix 方法，并将返回的字典对象的 Key 和 Value 通过表格的形式呈现出来。

```
@model NameValueCollectionValueProvider
<html>
  <head>
    <title>指定前缀的 Key</title>
  </head>
  <body>
    <table>
      <tr><th colspan="2">foo</th></tr>
      @foreach (var item in Model.GetKeysFromPrefix("foo"))
      {
        <tr><td>@item.Key</td><td>@item.Value</td></tr>
      }

      <tr><th colspan="2">foo.Address</th></tr>
      @foreach (var item in Model.GetKeysFromPrefix("foo.Address"))
      {
        <tr><td>@item.Key</td><td>@item.Value</td></tr>
      }
    </table>
  </body>
</html>
```

运行该程序后会在浏览器上呈现出如图 5-2 所示的输出结果。可以看到对于针对指定前缀返回的字典对象，作为 Key 和 Value 的字符串的不同之处在于前者没有包含指定的前缀而后者包含。除此之外，字典对象包含的元素全部处于同一级别：将“foo”指定为前缀时返回的元素针对 Contact 的四个属性，但不包含下一级（Address 对象的四个属性）的元素。虽然 NameValueCollection 中并不包含一个名为“foo.Address”的元素，但是 ASP.NET MVC 依然会将其单独作为以“foo”为前缀的 Key。（S501）

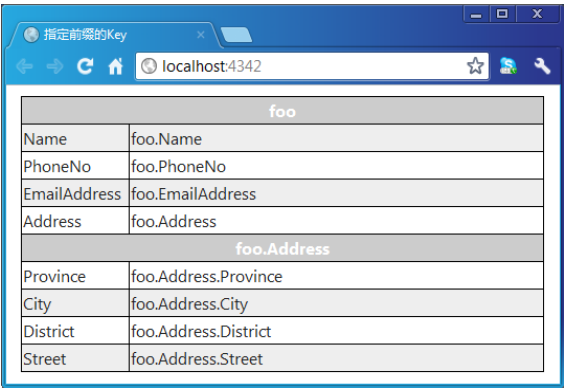


图 5-2 GetKeysFromPrefix 方法返回的字典对象的结构（1）

接下来我们采用类似的方式来演示基于索引的前缀，为此我们将 HomeController 的

Index 方法进行了如下的改写：作为数据源的 NameValueCollection 对象针对一个包含两个元素的 Contact 集合，为这个 Contact 集合对象提供数据的元素以字符“first”为前缀。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        NameValueCollection datasource = new NameValueCollection();

        datasource.Add("first[0].Name", "Foo");
        datasource.Add("first[0].PhoneNo", "123456789");
        datasource.Add("first[0].EmailAddress", "Foo@gmail.com");

        datasource.Add("first[1].Name", "Bar");
        datasource.Add("first[1].PhoneNo", "987654321");
        datasource.Add("first[1].EmailAddress", "Bar@gmail.com");

        NameValueCollectionValueProvider valueProvider =
            new NameValueCollectionValueProvider(datasource,
                CultureInfo.InvariantCulture);

        return View(valueProvider);
    }
}
```

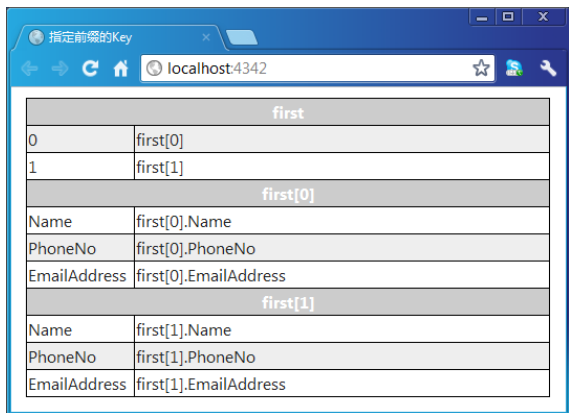
我们对 View 进行如下的改动，将三个表示前缀的字符串“first”、“first[0]”和“first[1]”作为参数调用 NameValueCollectionValueProvider 对象 GetKeysFromPrefix 方法，并将获取的相应字典的 Key 和 Value 呈现出来。

```
@model NameValueCollectionValueProvider
<html>
<head>
    <title>指定前缀的 Key</title>
</head>
<body>
    <table>
        <tr><th colspan="2">first</th></tr>
        @foreach (var item in Model.GetKeysFromPrefix("first"))
        {
            <tr><td>@item.Key</td><td>@item.Value</td></tr>
        }

        <tr><th colspan="2">first[0]</th></tr>
        @foreach (var item in Model.GetKeysFromPrefix("first[0]"))
        {
            <tr><td>@item.Key</td><td>@item.Value</td></tr>
        }

        <tr><th colspan="2">first[1]</th></tr>
        @foreach (var item in Model.GetKeysFromPrefix("first[1]"))
        {
            <tr><td>@item.Key</td><td>@item.Value</td></tr>
        }
    </table>
</body>
</html>
```

该程序执行之后会在浏览器中产生如图 5-3 所示的输出结果。如果我们将 “[” 和 “]” 视为和 “.” 一样的分割符，体现在方法 `GetKeysFromPrefix` 上针对索引作为前缀的规则与基于 “.” 前缀的规则其实没有本质的区别。（S502）



first	
0	first[0]
1	first[1]
first[0]	
Name	first[0].Name
PhoneNo	first[0].PhoneNo
EmailAddress	first[0].EmailAddress
first[1]	
Name	first[1].Name
PhoneNo	first[1].PhoneNo
EmailAddress	first[1].EmailAddress

图 5-3 `GetKeysFromPrefix` 方法返回的字典对象的结构（2）

FormValueProvider 与 QueryStringValueProvider

HTTP 请求提交的表单和请求查询字符串是 Model 绑定的两个主要的数据来源，针对它们的数据提供实现在两个具体的 `NameValueCollectionValueProvider` 中，它们分别是 `FormValueProvider` 和 `QueryStringValueProvider`，两个类型均定义在命名空间 `System.Web.Mvc` 下。如下面的代码片段所示，作为这两个 `NameValueCollectionValueProvider` 数据源容器的 `NameValueCollection` 对象分别是当前请求的表单（`HttpRequest` 的 `Form` 属性）和查询字符串集合（`HttpRequest` 的 `QueryString` 属性）。

```
public sealed class FormValueProvider : NameValueCollectionValueProvider
{
    public FormValueProvider(ControllerContext controllerContext)
        : base(controllerContext.RequestContext.HttpContext.Request.Form,
            CultureInfo.CurrentCulture)
    { }
}

public sealed class NameValueCollection : NameValueCollectionValueProvider
{
    public NameValueCollection(ControllerContext controllerContext)
        : base(controllerContext.RequestContext.HttpContext.Request.QueryString,
            CultureInfo.CurrentCulture)
    { }
}
```

5.2.2 DictionaryValueProvider

通过 `NameValueCollectionValueProvider` 提供的数据源将保存在一个 `NameValueCollection`

对象中，`DictionaryValueProvider` 自然将数据源存放在一个真正的字典对象之中。它们之间的不同之处在于 `NameValueCollection` 中的元素仅限于字符串，并且不对 `Key` 作唯一性约束；字典中的 `Key` 则是唯一的，`Value` 也不仅仅局限于字符串。

`DictionaryValueProvider` 的类型全名为 `System.Web.Mvc.DictionaryValueProvider<TValue>`，泛型参数 `TValue` 表示作为数据容器字典的 `Value` 类型。如下面的代码片段所示，它实现了 `IEnumerableValueProvider` 和 `IValueProvider` 接口，构造函数接受一个 `IDictionary<string, TValue>` 对象作为数据源容器。定义在 `DictionaryValueProvider<TValue>` 中所有方法的逻辑与定义在 `NameValueCollectionValueProvider` 中的同名方法基本一致。

```
public class DictionaryValueProvider<TValue> : IEnumerableValueProvider,
    IValueProvider
{
    public DictionaryValueProvider(IDictionary<string, TValue> dictionary,
        CultureInfo culture);
    public virtual bool ContainsPrefix(string prefix);
    public virtual IDictionary<string, string> GetKeysFromPrefix(string prefix);
    public virtual ValueProviderResult GetValue(string key);
}
```

RouteDataValueProvider

通过 URL 路由系统解析请求地址得到的路由数据可以作为 Model 绑定的数据来源，这样的数据通过类型为 `System.Web.Mvc.RouteDataValueProvider` 的对象来提供。如下面的代码片段所示，`RouteDataValueProvider` 继承自 `DictionaryValueProvider<Object>`，它在构造函数中获取保存于当前 `ControllerContext` 中表示路由数据的 `RouteData` 对象，并将其 `Values` 属性表示的字典作为自身的数据容器。

```
public sealed class RouteDataValueProvider : DictionaryValueProvider<object>
{
    public RouteDataValueProvider(ControllerContext controllerContext) :
        base(controllerContext.RouteData.Values, CultureInfo.InvariantCulture)
    {
    }
}
```

HttpFileCollectionValueProvider

可以通过类型为 `file` 的 `<input>` 元素进行文件的上传。在表示 HTTP 请求的 `HttpRequestBase` 对象中，上传文件通过只读属性 `Files` 表示。如下面的代码片段所示，该属性类型为 `HttpFileCollectionBase`，是一个元素类型为 `HttpPostedFileBase` 的集合。

```
public abstract class HttpRequestBase
{
    public virtual HttpFileCollectionBase Files { get; }
}

public abstract class HttpFileCollectionBase : NameObjectCollectionBase,
```

```

    ICollection, IEnumerable
{
    public virtual string[] AllKeys { get; }
    public override int Count { get; }

    public virtual HttpPostedFileBase this[int index] { get; }
    public virtual HttpPostedFileBase this[string name] { get; }
}

public abstract class HttpPostedFileBase
{
    public virtual void SaveAs(string filename);

    public virtual int ContentLength { get; }
    public virtual string ContentType { get; }
    public virtual string FileName { get; }
    public virtual Stream InputStream { get; }
}

```

对于处理上传文件的 Action 来说,通常定义一个或者多个类型为 `HttpPostedFileBase` (处理单个上传文件) 或者 `IEnumerable<HttpPostedFileBase>` (处理一组上传文件) 的参数来接收上传的文件。针对参数类型为 `HttpPostedFileBase` 的 Model 绑定利用一个 `System.Web.Mvc.HttpFileCollectionValueProvider` 对象来提供所需的数据。如下面的代码片段所示, `HttpFileCollectionValueProvider` 继承自 `DictionaryValueProvider<HttpPostedFileBase[]>`, 也就是说作为数据源容器的是一个值类型为 `HttpPostedFileBase` 数组 (不是 `HttpPostedFileBase`) 的字典。

```

public sealed class HttpFileCollectionValueProvider :
    DictionaryValueProvider<HttpPostedFileBase[]>
{
    public HttpFileCollectionValueProvider(ControllerContext controllerContext);
}

```

当我们根据当前 `ControllerContext` 构建一个 `HttpFileCollectionValueProvider` 的时候, ASP.NET MVC 会从当前 HTTP 请求的 `Files` 属性中获取所有的 `HttpPostedFileBase` 对象。多个 `HttpPostedFileBase` 可以共享相同的名称, 作为数据源容器的字典将 `HttpPostedFileBase` 的名称作为 Key, 具有相同名称的一个或者多个 `HttpPostedFileBase` 对象构成一个数组作为对应的 Value。

ChildActionValueProvider

子 Action 和普通 Action 的不同之处在于它不能用于响应来自客户端的请求, 只是在某个 View 中被调用以生成某个部分的 HTML。如下面的代码片段所示, `HtmlHelper` 具有一系列名为 Action 的扩展方法重载用于调用指定 Action 以生成相应的 HTML。

```

public static class ChildActionExtensions
{
    //其他成员
    public static MvcHtmlString Action(this HtmlHelper htmlHelper,
        string actionName);
}

```



```

public static MvcHtmlString Action(this HtmlHelper htmlHelper,
    string actionName, object routeValues);
public static MvcHtmlString Action(this HtmlHelper htmlHelper,
    string actionName, string controllerName);
public static MvcHtmlString Action(this HtmlHelper htmlHelper,
    string actionName, RouteValueDictionary routeValues);
public static MvcHtmlString Action(this HtmlHelper htmlHelper,
    string actionName, string controllerName, object routeValues);
public static MvcHtmlString Action(this HtmlHelper htmlHelper,
    string actionName, string controllerName,
    RouteValueDictionary routeValues);
}

```

作为子 `Action` 方法参数的数据来源与普通 `Action` 方法有所不同，`Model` 绑定过程中具体的数据提供由一个类型为 `System.Web.Mvc.ChildActionValueProvider` 的对象来完成。如下面的代码片段所示，`ChildActionValueProvider` 依然是 `DictionaryValueProvider<Object>` 的继承者，那么在作为数据源容器的字典中，具体的 `Key` 和 `Value` 究竟是怎样一个对象呢？

```

public sealed class ChildActionValueProvider :
    DictionaryValueProvider<object>
{
    public ChildActionValueProvider(ControllerContext controllerContext);
    public override ValueProviderResult GetValue(string key);
}

```

当我们针对指定的 `ControllerContext` 创建一个 `ChildActionValueProvider` 对象时，`ControllerContext` 中表示路由数据的 `RouteData` 对象会被提取出来，其 `Values` 属性表示的 `RouteValueDictionary` 对象会作为 `ChildActionValueProvider` 的数据容器字典，这可以从 `ChildActionValueProvider` 的构造函数的定义看出来。

```

public sealed class ChildActionValueProvider :
    DictionaryValueProvider<object>
{
    //其他成员
    public ChildActionValueProvider(ControllerContext controllerContext) :
        base(controllerContext.RouteData.Values, CultureInfo.InvariantCulture)
    {
    }
}

```

但是 `ChildActionValueProvider` 的 `GetValue` 方法针对给定的 `Key` 获取的值却并不是简单地来源于原始的路由数据，不然 `ChildActionValueProvider` 就和 `RouteDataValueProvider` 没有什么分别了。实际上 `ChildActionValueProvider` 的 `GetValue` 方法获取的值来源于调用 `HtmlHelper` 的扩展方法 `Action` 时，通过参数 `routeValues` 指定的 `RouteValueDictionary` 对象。

现在来简单介绍一下 `ChildActionValueProvider` 的 `GetValue` 方法的实现逻辑。如下面的代码片段所示，`ChildActionValueProvider` 具有一个字符串类型的静态字段 `_childActionValuesKey`。当该类型第一次被加载时（静态构造函数被调用），该字段被初始化成一个 `GUID`。

```
public sealed class ChildActionValueProvider :
DictionaryValueProvider<object>
{
    //其他成员
    private static string _childActionValuesKey = Guid.NewGuid().ToString();
}
```

当我们通过 `HtmlHelper` 的扩展方法 `Action` 调用某个指定的子 `Action` 时，如果参数 `routeValues` 指定的 `RouteValueDictionary` 不为空，`HtmlHelper` 会据此创建一个 `DictionaryValueProvider<Object>` 对象，并将这个对象添加到通过 `routeValues` 参数表示的原始的 `RouteValueDictionary` 对象中，对应的 `Key` 就是 `ChildActionValueProvider` 的静态属性 `_childActionValuesKey` 所表示的 GUID。

这个 `RouteValueDictionary` 被进一步封装成表示请求上下文的 `RequestContext` 对象，随后被调子 `Action` 所在的 `Controller` 会在该请求上下文中被激活，在 `Controller` 激活过程中表示 `ControllerContext` 的 `ControllerContext` 被创建出来，毫无疑问它包含了之前创建的 `RouteValueDictionary` 对象。当我们针对当前 `ControllerContext` 创建 `ChildActionValueProvider` 的时候，作为数据源的 `RouteValueDictionary` 就是这么一个对象。

```
@Html.Action("XxxChildAction", new {Foo=123, Bar = 456, Baz=789})
```

举个例子，假设我们在某个 `View` 中采用如上代码调用当前 `Controller` 的子 `Action` 方法“`XxxChildAction`”，并指定相应的路由变量(`Foo`、`Bar` 和 `Baz`)。最终作为 `ChildActionValueProvider` 数据源的 `Dictionary<string, object>` 将具有如图 5-4 所示的结构，其中左边部分为 `Key`，右边部分为 `Value`。

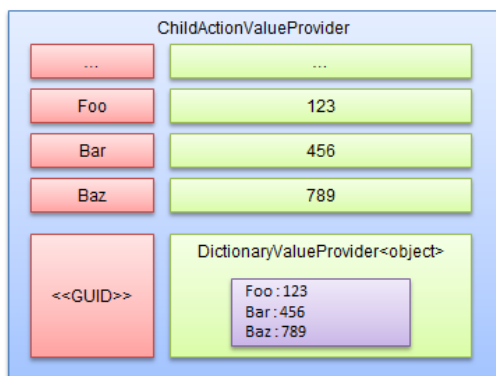


图 5-4 ChildActionValueProvider 数据容器的结构

当调用 `ChildActionValueProvider` 的 `GetValue` 方法获取指定 `Key` 的值时，实际上它并不会直接根据指定的 `Key` 去获取对应的值，而是根据通过其静态字段 `_childActionValuesKey` 值去获取对应的 `DictionaryValueProvider<object>` 对象，然后再调用该对象的 `GetValue` 根据指定的 `Key` 去获得相应的值。

实例演示：ChildActionValueProvider 的值提供机制（S503）

为了印证上面介绍的关于 ChildActionValueProvider 的数据提供机制，我们来演示一个简单的实例。在进行演示之前需要作一下简单说明：对于 DictionaryValueProvider<TValue>对象来说，作为数据容器的字典通过 _values 字段表示，如下面的代码片段所示，这是一个只读字段，所以我们为 DictionaryValueProvider<TValue>定义了如下一个 GetDataSource 扩展方法将该字段的值提取出来。

```
public class DictionaryValueProvider<TValue> : IEnumerableValueProvider,
    IValueProvider
{
    //其他成员
    private readonly Dictionary<string, ValueProviderResult> _values;
}

public static class DictionaryValueProviderExtensions
{
    public static Dictionary<string, ValueProviderResult> GetDataSource<TValue>(
        this DictionaryValueProvider<TValue> valueProvider)
    {
        FieldInfo valuesField = typeof(DictionaryValueProvider<TValue>)
            .GetField("_values", BindingFlags.Instance | BindingFlags.NonPublic);
        return (Dictionary<string,
            ValueProviderResult>)valuesField.GetValue(valueProvider);
    }
}
```

我们在一个 ASP.NET MVC 应用中定义了如下一个 HomeController。默认的 Action 方法 Index 将对应的 View 呈现出来，而另一个 Action 方法 DataOfChildActionValueProvider 则根据当前 ControllerContext 创建一个 ChildActionValueProvider 对象，并将其作为 Model 呈现在对应的 View 中。DataOfChildActionValueProvider 方法在返回 ViewResult 之前，我们将存在于当前 ControllerContext 中的三个路由变量（Foo、Bar 和 Baz）的值进行了相应的修改。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    public ActionResult DataOfChildActionValueProvider()
    {
        ControllerContext.RouteData.Values["Foo"] = "abc";
        ControllerContext.RouteData.Values["Bar"] = "ijk";
        ControllerContext.RouteData.Values["Baz"] = "xyz";

        ChildActionValueProvider valueProvider =
```

```

        new ChildActionValueProvider(ControllerContext);
    }
    return View(valueProvider);
}
}

```

如下所示的是 Action 方法 Index 对应 View 的定义，在该 View 中我们调用 HtmlHelper 的扩展方法 Action 以子 Action 的方式调用 DataOfChildActionValueProvider，并将生成的内容作为主体部分的 HTML。在进行子 Action 调用的同时，还指定了在 DataOfChildActionValueProvider 方法中被篡改的路由变量。

```

<html>
  <head>
    <title>ChildActionValueProvider 的数据结构</title>
  </head>
  <body>
    @Html.Action("DataOfChildActionValueProvider",
      new { Foo = 123, Bar = 456, Baz = 789 })
  </body>
</html>

```

Action 方法 DataOfChildActionValueProvider 对应 View 的整个内容定义如下，可以看到这是一个 Model 类型为 ChildActionValueProvider 的强类型 View。在该 View 中调用 DictionaryValueProvider<TValue> 的扩展方法 GetDataSource 获取作为 ChildActionValueProvider 对象数据容器的 Dictionary<string, ValueProviderResult> 对象，并将 Key 和 Value 通过表格的形式呈现出来。如果某个元素的值是一个 DictionaryValueProvider<object> 对象，同样通过调用扩展方法 GetDataSource 得到作为它数据容器的字典对象，并将其结构呈现出来。

```

@model ChildActionValueProvider
@{
    var dictionary1 = this.Model.GetDataSource();
}
<table>
  <tr>
    <th>Key</th><th colspan="2">Value</th>
  </tr>
  @foreach (var item1 in dictionary1)
  {
      DictionaryValueProvider<object> valueProvider =
          item1.Value.RawValue as DictionaryValueProvider<object>;
      if (null == valueProvider)
      {
          <tr>
            <td>@item1.Key</td><td colspan="2">@item1.Value.RawValue</td>
          </tr>
        }
      else
      {
          var dictionary2 = valueProvider.GetDataSource();
          <tr>

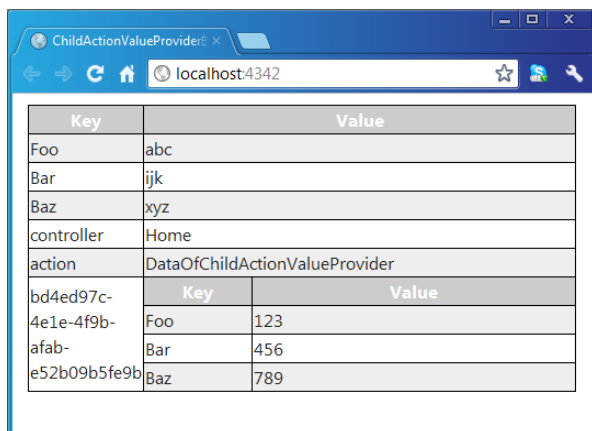
```

```

        <td rowspan="@dictionary2.Count + 1">@item1.Key</td>
        <th>Key</th><th>Value</th>
    </tr>
    foreach(var item2 in dictionary2)
    {
        <tr><td>@item2.Key</td><td>@item2.Value.RawValue</td></tr>
    }
}
</table>

```

该程序运行之后会在浏览器中呈现出如图 5-5 所示的输出结果，它正好反映了 `ChildActionValueProvider` 数据容器具有前面我们介绍过的数据结构。对于这个例子来说，如果我们调用 `ChildActionValueProvider` 的 `GetValue` 方法试图获取 `Foo`、`Bar` 和 `Baz` 三个路由变量，得到的值应该是 123、456 和 789，而不是“abc”、“ijk”和“xyz”。如果我们将“controller”或者“action”作为 `GetValue` 方法的参数，则会直接返回 Null。



Key	Value								
Foo	abc								
Bar	ijk								
Baz	xyz								
controller	Home								
action	DataOfChildActionValueProvider								
bd4ed97c-4e1e-4f9b-afab-e52b09b5fe9b	<table border="1"> <thead> <tr> <th>Key</th><th>Value</th></tr> </thead> <tbody> <tr> <td>Foo</td><td>123</td></tr> <tr> <td>Bar</td><td>456</td></tr> <tr> <td>Baz</td><td>789</td></tr> </tbody> </table>	Key	Value	Foo	123	Bar	456	Baz	789
Key	Value								
Foo	123								
Bar	456								
Baz	789								

图 5-5 `ChildActionValueProvider` 数据容器的结构

ValueProviderCollection

`System.Web.Mvc.ValueProviderCollection` 表示一个元素类型为 `IValueProvider` 的集合，除此之外，它本身也是一个 `ValueProvider`。如下面的代码片段所示，`ValueProviderCollection` 不仅仅实现了 `IValueProvider` 接口，还实现了 `IUnvalidatedValueProvider` 和 `IEnumerableValueProvider` 接口。

```

public class ValueProviderCollection : Collection<IValueProvider>,
    IUnvalidatedValueProvider, IEnumerableValueProvider, IValueProvider
{
    public ValueProviderCollection();
    public ValueProviderCollection(IList<IValueProvider> list);

    public virtual bool ContainsPrefix(string prefix);
}

```

```

public virtual IDictionary<string, string> GetKeysFromPrefix(string prefix);
public virtual ValueProviderResult GetValue(string key);
public virtual ValueProviderResult GetValue(string key, bool skipValidation);
}

```

当 `ValueProviderCollection` 的 `GetValue` 方法被调用的时候，它会遍历整个集合并传入指定的参数调用 `ValueProvider` 的同名方法，直到返回一个具体的 `ValueProviderResult` 对象。如果调用集合中所有的 `ValueProvider` 的 `GetValue` 方法均返回 `Null`，那么最终的返回值就是 `Null`。

用于获取指定前缀的 `Key` 的 `GetKeysFromPrefix` 方法具有与 `GetValue` 类似的实现逻辑，唯一的不同之处在于，如果调用所有 `IEnumerableValueProvider` 的 `GetKeysFromPrefix` 方法均返回一个空的字典对象，那么最终的结果也是一个空的字典。至于另一个方法 `ContainsPrefix`，如果有任何一个 `ValueProvider` 的 `ContainsPrefix` 方法返回 `True`，则 `ValueProviderCollection` 的 `ContainsPrefix` 也返回 `True`。

5.2.3 ValueProviderFactory

`ValueProviderFactory` 是创建 `ValueProvider` 的工厂，它们继承自具有如下定义的抽象类 `System.Web.Mvc.ValueProviderFactory`，唯一的抽象方法 `GetValueProvider` 会根据当前 `ControllerContext` 创建相应的 `ValueProvider` 对象。

```

public abstract class ValueProviderFactory
{
    public abstract IValueProvider GetValueProvider(
        ControllerContext controllerContext);
}

```

在 `System.Web.Mvc` 命名空间下，ASP.NET MVC 定义了一系列具体的 `ValueProviderFactory` 类型。前面介绍的两个 `NameValueCollectionValueProvider`（`FormValueProvider` 和 `QueryStringValueProvider`）通过对应的 `FormValueProviderFactory` 和 `QueryStringValueProviderFactory` 创建，而三个具体的 `DictionaryValueProvider`（`RouteDataValueProvider`、`HttpFileCollectionValueProvider` 和 `ChildActionValueProvider`）对应的 `ValueProviderFactory` 类型分别是 `RouteDataValueProviderFactory`、`HttpFileCollectionValueProviderFactory` 和 `ChildActionValueProviderFactory`。除此之外，ASP.NET MVC 还定义了一个 `JsonValueProviderFactory`，它会将 JSON 格式的请求内容转换成字典对象，并据此创建一个 `DictionaryValueProvider` 对象。

5.2.4 ValueProviderFactories

`ValueProviderFactory` 的注册通过静态类型 `System.Web.Mvc.ValueProviderFactories` 来实现。如下面的代码片段所示，`ValueProviderFactory` 具有一个静态只读属性 `Factories`，它返回

一个表示 `ValueProviderFactory` 集合的 `ValueProviderFactoryCollection` 对象。

```
public static class ValueProviderFactories
{
    public static ValueProviderFactoryCollection Factories { get; }
}

public class ValueProviderFactoryCollection : Collection<ValueProviderFactory>
{
    public ValueProviderFactoryCollection();
    public ValueProviderFactoryCollection(IList<ValueProviderFactory> list);
    public IValueProvider GetValueProvider(ControllerContext controllerContext);
}
```

当执行 `ValueProviderFactoryCollection` 的 `GetValueProvider` 方法的时候，集合中每一个 `ValueProviderFactory` 对象的同名方法会被执行，它们创建的 `ValueProvider` 包含在最终返回的 `ValueProviderFactoryCollection` 对象中。`ValueProviderFactory` 在 `ValueProviderFactoryCollection` 集合中的先后次序决定了创建的 `ValueProvider` 在返回的 `ValueProviderCollection` 对象中的次序，而这个次序决定了使用的优先级。

如下面的代码片段所示，`ValueProviderFactories` 类型被加载的时候会对静态属性 `Factories` 进行初始化，上面介绍的 6 种 `ValueProviderFactory` 会被创建并添加到 `Factories` 集合中。由于添加的顺序体现了相应 `ValueProvider` 的选择优先级，意味着如果具有相同的名称的请求数据同时存在于提交的表单和查询字符串中，前者会被选用。

```
public static class ValueProviderFactories
{
    private static readonly ValueProviderFactoryCollection _factories =
        new ValueProviderFactoryCollection {
            new ChildActionValueProviderFactory(),
            new FormValueProviderFactory(),
            new JsonValueProviderFactory(),
            new RouteDataValueProviderFactory(),
            new QueryStringValueProviderFactory(),
            new HttpFileCollectionValueProviderFactory() };

    public static ValueProviderFactoryCollection Factories
    {
        get { return _factories; }
    }
}
```

以 `ValueProvider` 为核心的数据提供系统中涉及到了三类组件/类型：`ValueProvider`、`ValueProviderFactory` 和 `ValueProviderFactories`。`ValueProvider` 通过 `ValueProviderFactory` 创建，而 `ValueProviderFactory` 通过 `ValueProviderFactories` 进行注册。图 5-6 所示的 UML 体现了三者之间的关系。

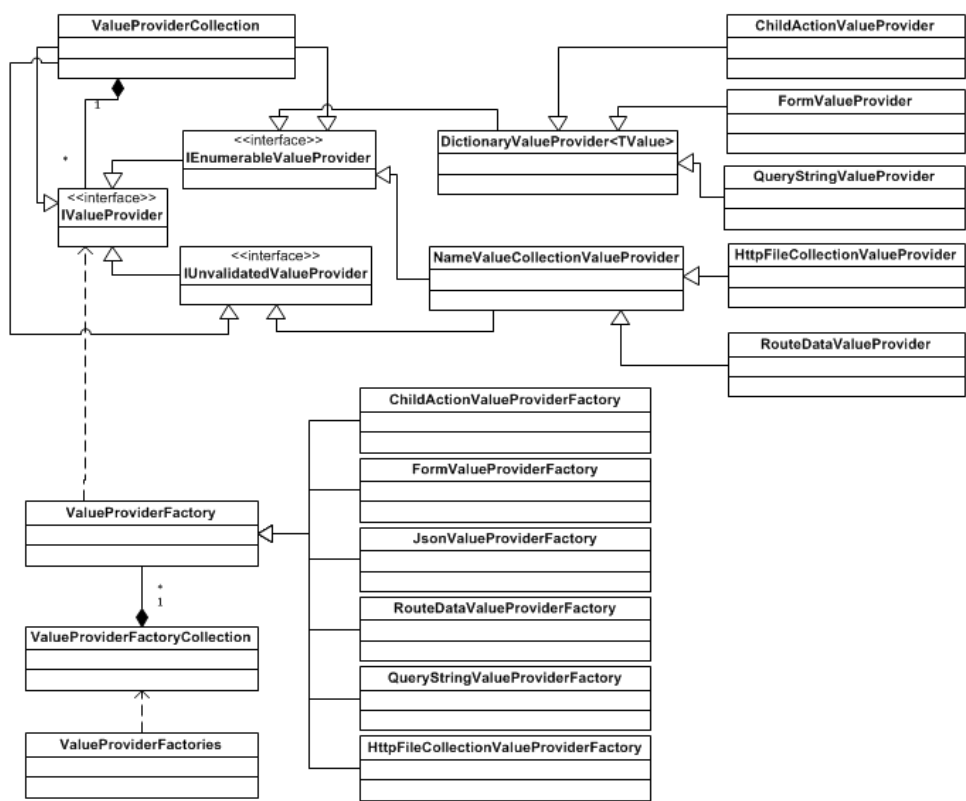


图 5-6 ValueProvider-ValueProviderFactory-ValueProviderFactories

实例演示：创建一个自定义 ValueProviderFactory（S504）

ASP.NET MVC 提供的 6 种 ValueProviderFactory 基本上可以满足绝大部分 Model 绑定需求，不过在一些特殊的场景下有可能需要自定义 ValueProviderFactory。在这个实例演示中，我们将创建一个以 HTTP 请求报头集合作为数据来源的 ValueProviderFactory。

将自定义的 ValueProviderFactory 命名为 **HttpHeaderValueProviderFactory**。如下面的代码片段所示，**HttpHeaderValueProviderFactory** 的定义非常简单，在重写的 **GetValueProvider** 方法中，根据指定的 **ControllerContext** 得到 HTTP 报头集合，并借此创建 **NameValueCollection** 对象，而最终返回的就是根据它创建的 **NameValueCollectionValueProvider**。为了与参数和属性命名相匹配，我们剔除了 HTTP 报头名称中包含的“-”字符。

```
public class HttpHeaderValueProviderFactory : ValueProviderFactory
{
    public override IValueProvider GetValueProvider(
        ControllerContext controllerContext)
    {
        NameValueCollection requestData = new NameValueCollection();
        var headers = controllerContext.RequestContext
            .HttpContext.Request.Headers;
```



```

        foreach (string key in headers.Keys)
        {
            requestData.Add(key.Replace("-", ""), headers[key]);
        }
        return new NameValueCollectionValueProvider(requestData,
            CultureInfo.InvariantCulture);
    }
}

```

在一个 ASP.NET MVCWeb 应用中定义了如下一个名为 CommonHttpHeaders 的数据类型，并在其中定义了 7 个属性，它们对应着 7 个相应的 HTTP 报头。

```

public class CommonHttpHeaders
{
    public string Connection { get; set; }
    public string Accept { get; set; }
    public string AcceptCharset { get; set; }
    public string AcceptEncoding { get; set; }
    public string AcceptLanguage { get; set; }
    public string Host { get; set; }
    public string UserAgent { get; set; }
}

```

然后定义了如下一个 HomeController，默认的 Action 方法 Index 具有一个类型为 CommonHttpHeaders 的参数，我们直接将该参数作为 Model 呈现在默认的 View 中。

```

public class HomeController : Controller
{
    public ActionResult Index(CommonHttpHeaders headers)
    {
        return View(headers);
    }
}

```

Action 方法 Index 对应的 View 具有如下的定义，可以看出这是 Model 类型为 CommonHttpHeaders 的强类型 View。在该 View 中，直接将包含在 CommonHttpHeaders 对象的 7 个 HTTP 报头的名称和值通过有关表格的形式呈现出来。

```

@model CommonHttpHeaders
<html>
<head>
    <title>HTTP 报头列表</title>
</head>
<body>
    <table>
        <tr><th>Name</th><th>Value</th></tr>
        <tr><td>Accept</td><td>@Model.Accept</td></tr>
        <tr><td>AcceptCharset</td><td>@Model.AcceptCharset</td></tr>
        <tr><td>AcceptEncoding</td><td>@Model.AcceptEncoding</td></tr>
        <tr><td>AcceptLanguage</td><td>@Model.AcceptLanguage</td></tr>
        <tr><td>Connection</td><td>@Model.Connection</td></tr>
        <tr><td>Host</td><td>@Model.Host</td></tr>
        <tr><td>UserAgent</td><td>@Model.UserAgent</td></tr>
    </table>
</body>
</html>

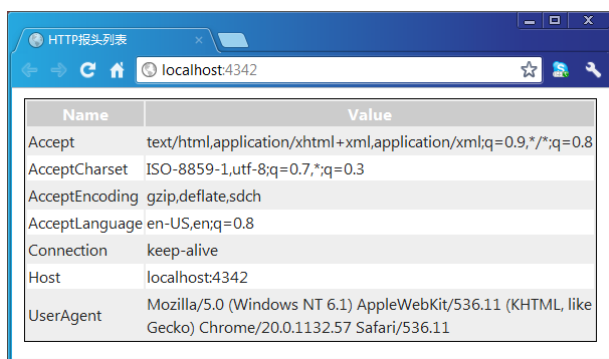
```

为了让我们自定义的 HttpHeaderValueProviderFactory 能够参与到 Model 绑定过程中以提

供目标 Action 的参数值（这里指 Action 方法 Index 中类型为 CommonHttpHeaders 的参数 headers），我们在 Global.asax 中采用如下的代码对这个自定义 ValueProviderFactory 进行注册。

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        ValueProviderFactories.Factories.Add(
            new HttpHeaderValueProviderFactory());
    }
}
```

该程序运行之后会在浏览器中呈现出如图 5-7 所示的输出结果，输出的正是当前请求的 7 个 HTTP 报头。



Name	Value
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
AcceptCharset	ISO-8859-1,utf-8;q=0.7,*;q=0.3
AcceptEncoding	gzip,deflate,sdch
AcceptLanguage	en-US,en;q=0.8
Connection	keep-alive
Host	localhost:4342
UserAgent	Mozilla/5.0 (Windows NT 6.1) AppleWebKit/536.11 (KHTML, like Gecko) Chrome/20.0.1132.57 Safari/536.11

图 5-7 自定义 HttpHeaderValueProviderFactory 实现的针对 HTTP 报头的值提供机制

5.3 ModelBinder

Model 的绑定体现在从当前请求提取相应的数据并生成相应的对象作为调用目标 Action 方法的参数列表，Model 绑定根据用以描述参数的元数据来进行。通过前面的介绍我们知道 Action 方法的参数元数据通过 ParameterDescriptor 来描述，通过它的属性的 BindingInfo 表示的 ParameterBindingInfo 对象具有一个名为 ModelBinder 的组件用于真正实现 Model 绑定。ModelBinder 是整个 Model 绑定系统的核心，我们先来认识下这个重要的组件。

5.3.1 ModelBinder 与 ModelBinderProvider

真正实现 Model 绑定的 ModelBinder 实现了接口 System.Web.Mvc.IModelBinder。如下面的代码片段所示，IModelBinder 接口具有唯一一个名为 BindModel 的方法，针对某个参数的绑定就实现在这个方法中。

```
public interface IModelBinder
{
    object BindModel(ControllerContext controllerContext,
```

```

        ModelBindingContext bindingContext);
    }

```

`IModelBinder` 的 `BindModel` 方法具有两个基于上下文的参数，一个是表示当前的 `ControllerContext`，另一个则是通过 `System.Web.ModelBinding.ModelBindingContext` 类型表示的 `Model` 绑定上下文。在 `Controller` 初始化的时候，`ControllerContext` 已经被创建出来，如果我们能够创建出针对当前 `Model` 上下文的 `ModelBindingContext` 对象就能够调用提供的 `ModelBinder` 生成相应的参数值。关于 `ModelBindingContext` 的创建会在后面进行单独介绍，在这之前先来了解一下 `ModelBinder` 的提供机制。

CustomModelBinderAttribute 与 ModelBinderAttribute

如果某种类型的 `ModelBinder` 显式绑定到 `Action` 方法的某个参数上，它会被优先选择用于针对该参数的 `Model` 绑定，参数与 `ModelBinder` 类型之间的绑定可以通过特性 `System.Web.Mvc.CustomModelBinderAttribute` 来完成。如下面的代码片段所示，这是一个抽象类，它唯一的抽象方法 `GetBinder` 用于获取相应的 `ModelBinder` 对象。

```

[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Interface |
    AttributeTargets.Enum | AttributeTargets.Struct | AttributeTargets.Class,
    AllowMultiple=false, Inherited=false)]
public abstract class CustomModelBinderAttribute : Attribute
{
    public abstract IModelBinder GetBinder();
}

```

`CustomModelBinderAttribute` 具有一个唯一的继承者 `System.Web.Mvc.ModelBinderAttribute`，可以通过它定制 `Model` 绑定采用的 `ModelBinder` 类型。根据如下所示的应用在类型上的 `AttributeUsageAttribute` 特性的定义，该特性不仅可以应用在参数上，也可以应用到类型（接口、枚举、结构和类）上，这意味我们既可以将其应用在 `Action` 方法的某个参数上，也可以将其应用在参数对应的数据类型上，但是 `ParameterDescriptor` 只会解析应用在参数上的特性。

```

[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Interface |
    AttributeTargets.Enum | AttributeTargets.Struct | AttributeTargets.Class,
    AllowMultiple=false, Inherited=false)]
public sealed class ModelBinderAttribute : CustomModelBinderAttribute
{
    public ModelBinderAttribute(Type binderType);
    public override IModelBinder GetBinder();

    public Type BinderType { [CompilerGenerated] get; }
}

```

为了演示 `ModelBinderAttribute` 特性对 `ParameterDescriptor` 的影响，我们来进行一个简单的实例演示。在一个 ASP.NET MVC 应用中定义了如下几个类型，`FooModelBinder` 和 `BarModelBinder` 是自定义的 `ModelBinder`，而 `Foo`、`Bar` 和 `Baz` 是三个作为 `Action` 方法参数的数据类型，其中类型 `Bar` 上应用了 `ModelBinderAttribute` 特性并将 `ModelBinder` 类型设置为 `BarModelBinder`。

```

public class FooModelBinder : IModelBinder
{
    public object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        throw new NotImplementedException();
    }
}

public class BarModelBinder : IModelBinder
{
    public object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        throw new NotImplementedException();
    }
}

public class Foo { }

[ModelBinder(typeof(BarModelBinder))]
public class Bar { }

public class Baz { }

```

我们定义了如下一个 HomeController，DemoAction 是一个空 Action 方法，它具有针对上面定义的三种数据类型（Foo、Bar 和 Baz）的参数，其中参数 foo 上应用了 ModelBinderAttribute 特性并将 ModelBinder 类型设置为 FooModelBinder。在 Action 方法 Index 中我们创建了一个用于描述 DemoAction 的 ActionDescriptor 对象，并将其作为 Model 呈现在默认的 View 中。

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        ControllerDescriptor controllerDescriptor =
            new ReflectedControllerDescriptor(typeof(HomeController));
        ActionDescriptor actionDescriptor = controllerDescriptor
            .FindAction(ControllerContext, "DemoAction");
        return View(actionDescriptor);
    }

    public void DemoAction(
        [ModelBinder(typeof(FooModelBinder))]
        Foo foo,
        Bar bar,
        Baz baz)
    { }
}

```

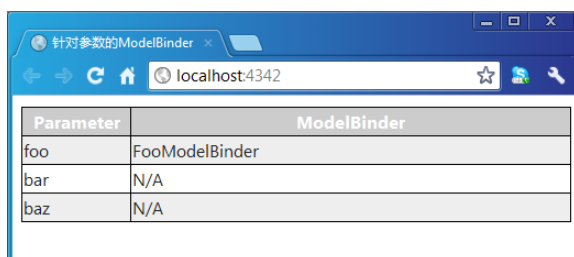
如下所示的是 Action 方法 Index 对应 View 的定义，可以看出这是一个 Model 类型为 ActionDescriptor 的强类型 View。在该 View 中，我们调用作为 Model 的 ActionDescriptor 对象的 GetParameters 方法得到用于描述所有参数的 ParameterDescriptor 列表，并将参数名称和对应的 ModelBinder 类型通过表格的形式呈现出来。

```

@model ActionDescriptor
<html>
  <head>
    <title>针对参数的 ModelBinder</title>
  </head>
  <body>
    <table>
      <tr><th>Parameter</th><th>ModelBinder</th></tr>
      @foreach (var parameter in Model.GetParameters())
      {
        string binderType = "N/A";
        IModelBinder binder = parameter.BindingInfo.Binder;
        if (null != binder)
        {
          binderType = binder.GetType().Name;
        }
        <tr><td>@parameter.ParameterName</td><td>@binderType</td></tr>
      }
    </table>
  </body>
</html>

```

该实例程序运行之后会在浏览器中呈现出图 5-8 所示的输出结果。可以清楚地看到应用到数据类型 `Bar` 上的 `ModelBinderAttribute` 特性设置的 `ModelBinder` 类型 (`BarModelBinder`) 并不会体现在用于描述参数的 `ParameterDescriptor` 对象上。(S505)



Parameter	ModelBinder
foo	FooModelBinder
bar	N/A
baz	N/A

图 5-8 针对参数的 ModelBinder (1)

ModelBinders

如果我们不曾利用 `ModelBinderAttribute` 特性将某个 `ModelBinder` 类型绑定到 `Action` 方法的某个参数上，默认采用的 `ModelBinder` 是通过静态类型 `System.Web.Mvc.ModelBinders` 来提供的。如下面的代码片段所示，`ModelBinders` 具有一个静态只读属性 `Binders`，它表示当前注册的 `ModelBinder` 列表，其类型为 `System.Web.Mvc.ModelBinderDictionary`。

```

public static class ModelBinders
{
    public static ModelBinderDictionary Binders { get; }
}

public class ModelBinderDictionary :
    IDictionary<Type, IModelBinder>,
    ICollection<KeyValuePair<Type, IModelBinder>>,
    IEnumerable<KeyValuePair<Type, IModelBinder>>,

```

```

IEnumerable
{
    //其他成员
    public IModelBinder GetBinder(Type modelType);
    public virtual IModelBinder GetBinder(Type modelType,
        bool fallbackToDefault);
}

```

ModelBinderDictionary 是一个以数据类型为 Key, ModelBinder 对象为 Value 的字典, 它定义了数据类型与 ModelBinder 之间的匹配关系。ModelBinderDictionary 的两个 GetBinder 方法重载用于获取针对某个数据类型的 ModelBinder 对象, 布尔类型的参数 fallbackToDefault 表示在匹配关系不存在的情况下是否采用默认的 ModelBinder 作为后备, 基于默认 ModelBinder 的后备机制会在第一个 GetBinder 方法重载中采用。这里默认 ModelBinder 类型为 System.Web.Mvc.DefaultModelBinder。

在针对某个参数进行 Model 绑定的时候, 需要先获取相应的 ModelBinder 对象。如果对应的 ParameterDescriptor 的 ModelBinder 不存在, 则通过 ModelBinders 的静态属性 Binders 获取表示当前注册的 ModelBinder 列表的 ModelBinderDictionary 对象, 并调用 GetBinder 方法获取指定参数类型的 ModelBinder 对象。在针对参数类型对 ModelBinder 进行解析的时候, 会提取应用在数据类型上的 ModelBinderAttribute 特性, 通过该特性设置的 ModelBinder 类型将用于创建绑定该参数的 ModelBinder 对象。

我们根据 ModelBinder 的提供机制对前面演示实例中用于呈现 Action 参数对应 ModelBinder 类型的 View 进行如下的修改, 在获取某个参数对应 ModelBinder 的时候, 先判断作为参数描述对象的 ParameterDescriptor 中是否包含对应的 ModelBinder, 如果不存在则利用静态 ModelBinders 根据参数类型获取对应 ModelBinder。

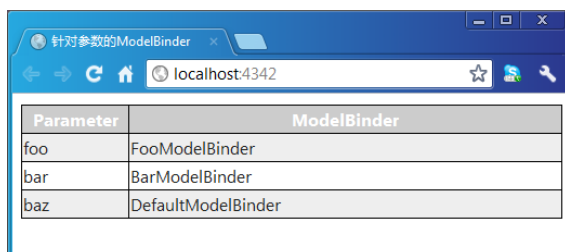
```

@model ActionDescriptor
<html>
    <head>
        <title>针对参数的 ModelBinder</title>
    </head>
    <body>
        <table>
            <tr><th>Parameter</th><th>ModelBinder</th></tr>
            @foreach (var parameter in Model.GetParameters())
            {
                string binderType = "N/A";
                IModelBinder binder = parameter.BindingInfo.Binder ??
                    ModelBinders.Binders.GetBinder(parameter.ParameterType);
                if (null != binder)
                {
                    binderType = binder.GetType().Name;
                }
                <tr><td>@parameter.ParameterName</td><td>@binderType</td></tr>
            }
        </table>
    </body>
</html>

```

再次运行我们的程序会在浏览器中呈现出如图 5-9 所示的输出结果。由于

FooModelBinder 和 BarModelBinder 通过 ModelBinderAttribute 特性分别应用到参数和参数类型上，所以它们将被用于对应参数（foo 和 bar）的 Model 绑定，而另一个参数 baz 则会采用默认的 DefaultModelBinder 进行绑定。（S506）



Parameter	ModelBinder
foo	FooModelBinder
bar	BarModelBinder
baz	DefaultModelBinder

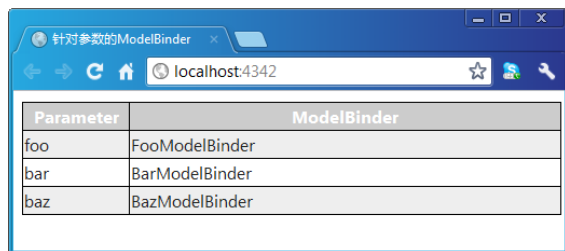
图 5-9 针对参数的 ModelBinder（2）

如果我们希望注册另一个 ModelBinder 来完成针对数据类型 Baz 的 Model 绑定，可以直接通过 ModelBinders 的静态属性 Binders 进行注册。在上面这个演示实例中，我们定义了如下一个 BazModelBinder，并在 Global.asax 中通过如下的代码将其注册到数据类型 Baz 上。

```
public class BazModelBinder : IModelBinder
{
    public object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        throw new NotImplementedException();
    }
}

public class MvcApplication : System.Web.HttpApplication
{
    //其他成员
    protected void Application_Start()
    {
        //其他操作
        ModelBinders.Binders.Add(typeof(Baz), new BazModelBinder());
    }
}
```

再次运行我们的程序，在浏览器中会得到如图 5-10 所示的输出结果，从中可以清楚地看出我们注册的 BazModelBinder 被用于 baz 参数的 Model 绑定。（S507）



Parameter	ModelBinder
foo	FooModelBinder
bar	BarModelBinder
baz	BazModelBinder

图 5-10 针对参数的 ModelBinder（3）

ModelBinderProvider

ASP.NET MVC 的 ModelBinder 提供机制还涉及另一个重要的组件 ModelBinderProvider，它们实现了具有如下定义的 System.Web.Mvc.IModelBinderProvider 接口。该接口定义了唯一的 GetBinder 方法用于根据指定的数据类型获取相应的 ModelBinder 对象，不过在 ASP.NET MVC 并没有定义任何一个实现该接口的 ModelBinderProvider 类型。

```
public interface IModelBinderProvider
{
    IModelBinder GetBinder(Type modelType);
}
```

我们可以利用具有如下定义的静态类型 System.Web.Mvc.ModelBinderProviders 进行自定义 ModelBinderProvider 的注册。ModelBinderProviders 具有一个静态只读属性 BinderProviders，它返回一个表示 ModelBinderProvider 集合的 System.Web.Mvc.ModelBinderProviderCollection 对象，所谓 ModelBinderProvider 的注册体现在将自定义的 ModelBinderProvider 对象添加到该集合之中。

```
public static class ModelBinderProviders
{
    public static ModelBinderProviderCollection BinderProviders { get; }
}

public sealed class ModelBinderProviderCollection :
    Collection<IModelBinderProvider>
{
    //省略成员
}
```

通过 ModelBinderProviders 的静态属性 BinderProviders 表示的 ModelBinderProvider 列表最终被 ModelBinderDictionary 使用。如下面的代码片段所示，ModelBinderDictionary 除了具有一个定义数据类型与 ModelBinder 匹配关系的字典（_innerDictionary 字段）和一个默认 ModelBinder（_defaultBinder）之外，还具有一个 ModelBinderProvider 列表（_modelBinderProviders 字段）。当 ModelBinderDictionary 被创建的时候，通过 ModelBinderProviders 的静态属性 BinderProviders 表示的 ModelBinderProvider 列表会用于初始化 _modelBinderProviders 字段。

```
public class ModelBinderDictionary
{
    //其他成员
    private IModelBinder _defaultBinder;
    private readonly Dictionary<Type, IModelBinder> _innerDictionary;
    private ModelBinderProviderCollection _modelBinderProviders;
}
```

以 ModelBinder 为核心的 Model 绑定系统包含的组件，以及它们之间的关系基本上可以通过图 5-11 所示的 UML 来表示。

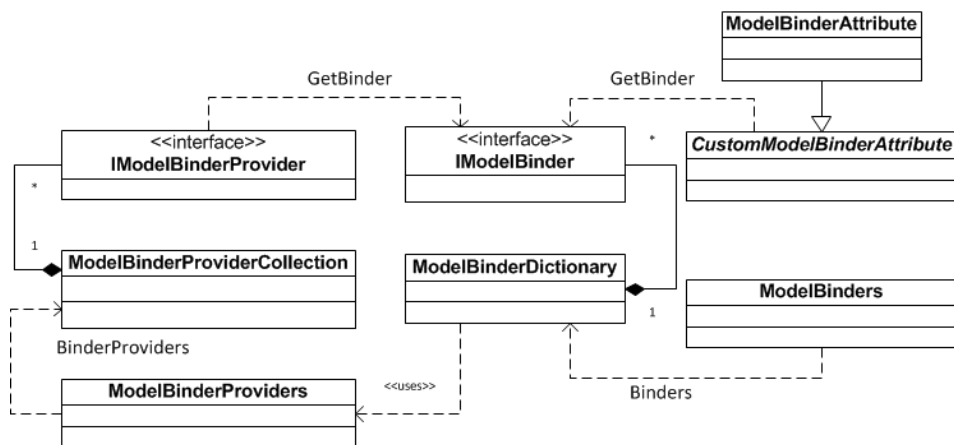


图 5-11 ModelBinder-ModelBinderProvider-CustomModelBinderAttribute

当调用 `ModelBinderProviderCollection` 的 `GetBinder` 方法获取指定数据类型的 `ModelBinder` 时，`_innerDictionary` 字段表示的 `ModelBinder` 字典会被优先选择。如果数据类型在该字典中找不到，则利用 `_modelBinderProviders` 字段表示的 `ModelBinderProvider` 列表来提供相应的 `ModelBinder`。只有在这两种 `ModelBinder` 提供方式均宣告失败的情况下才会选择通过 `_defaultBinder` 字段表示的默认 `ModelBinder`。也就是说，如果想为某个数据类型定制某种类型的 `ModelBinder`，按照选择优先级具有如下几种方式供选择。

- 将 `ModelBinderAttribute` 应用在 `Action` 方法的相应参数上并指定相应的 `ModelBinder` 类型，或者在参数上应用一个与之等效的自定义 `CustomModelBinderAttribute` 特性。
- 将 `ModelBinderAttribute` 应用在数据类型上并指定相应的 `ModelBinder` 类型，或者在数据类型上应用一个与之等效的自定义 `CustomModelBinderAttribute` 特性。
- 通过 `ModelBinders` 的静态属性 `Binders` 将某个具体的 `ModelBinder` 对象注册到相应的数据类型上（通过这种方式注册的 `ModelBinder` 与通过应用在数据类型上的 `ModelBinderAttribute` 特性注册的 `ModelBinder` 保存在相同的集合之中，其优先级取决于各自存在于集合中的位置）。
- 自定义 `ModelBinderProvider` 为某种数据类型提供对应的 `ModelBinder`，并添加到 `ModelBinderProviders` 的静态属性 `BinderProviders` 表示的 `ModelBinderProvider` 列表中。

前面三种方式的 `ModelBinder` 提供机制已经通过实例演示过了，现在来演示基于自定义 `ModelBinderProvider` 的 `ModelBinder` 提供机制。在前面的例子中我们为 `Foo`、`Bar` 和 `Baz` 这三种数据类型创建了相应的 `ModelBinder` (`FooModelBinder`、`BarModelBinder` 和 `BazModelBinder`)，现在创建如下一个自定义的 `MyModelBinderProvider` 将两者（数据类型和 `ModelBinder` 对象）进行关联。在 `Global.asax` 中按照如下的代码利用 `ModelBinderProviders` 对自定义的 `MyModelBinderProvider` 进行注册。

```
public class MyModelBinderProvider : IModelBinderProvider
{
```

```

public IModelBinder GetBinder(Type modelType)
{
    if (modelType == typeof(Foo))
    {
        return new FooModelBinder();
    }
    if (modelType == typeof(Bar))
    {
        return new BazModelBinder();
    }
    if (modelType == typeof(Baz))
    {
        return new BazModelBinder();
    }
    return null;
}
}

public class MvcApplication : System.Web.HttpApplication
{
    //其他成员
    protected void Application_Start()
    {
        //其他操作
        ModelBinderProviders.BinderProviders.Add(new MyModelBinderProvider());
    }
}

```

由于 `MyModelBinderProvider` 实现了针对 `Foo`、`Bar` 和 `Baz` 三种数据类型的 `ModelBinder` 的提供，所以我们需要将应用在 `Action` 方法参数（`foo`）和数据类型（`Bar`）上的 `ModelBinderAttribute` 特性删除。再次运行我们的程序，同样会在浏览器中呈现出如图 5-10 所示的输出结果。（S508）

5.3.2 ModelState 与 Model 绑定

`Model` 绑定除了利用 `ModelBinder` 为目标 `Action` 方法的执行提供参数之外，还会将相关的数据以 `ModelState` 的形式存储于当前 `Controller` 的 `ViewData` 中。如下面的代码片段所示，`Controller` 的基类 `ControllerBase` 中具有一个 `System.Web.Mvc.ViewDataDictionary` 类型的属性 `ViewData`。顾名思义，`ViewData` 就是 `Controller` 在进行 `View` 呈现过程中传递的数据。

```

public abstract class ControllerBase : IController
{
    //其他成员
    public ViewDataDictionary ViewData { get; set; }
}

public class ViewDataDictionary : IDictionary<string, object>,
    ICollection<KeyValuePair<string, object>>,
    IEnumerable<KeyValuePair<string, object>>,
    IEnumerable
{
    //其他成员
    public ModelStateDictionary ModelState { get; }
}

```

字典类型的 `ViewDataDictionary` 具有一个类型为 `System.Web.Mvc.ModelStateDictionary` 的属性 `ModelState`，这是一个 `Key` 和 `Value` 类型分别为 `String` 和 `System.Web.Mvc.ModelState` 的字典。在这里有一点需要引起读者注意：`ViewDataDictionary` 的 `ModelState` 属性类型不是 `ModelState`，而是 `ModelStateDictionary`。

```
[Serializable]
public class ModelStateDictionary : IDictionary<string, ModelState>,
    ICollection<KeyValuePair<string, ModelState>>,
    IEnumerable<KeyValuePair<string, ModelState>>,
    IEnumerable
{
}
```

`ModelState` 对象维护的 `Model` 状态具有两种类型，一类通过 `ValueProvider` 提供的 `ValueProviderResult` 对象，对应着属性 `Value`，另一类是通过 `System.Web.Mvc.ModelErrorCollection` 类型表示的错误。`ModelErrorCollection` 是一个元素类型为 `System.Web.Mvc.ModelError` 的集合，而 `ModelError` 通过属性 `ErrorMessage` 和 `Exception` 属性表述错误的消息和抛出的异常。如下面的代码片段所示，所有的这些类型都是可序列化的。

```
[Serializable]
public class ModelState
{
    public ModelErrorCollection Errors { get; }
    public ValueProviderResult Value { get; set; }
}

[Serializable]
public class ModelErrorCollection : Collection<ModelError>
{
    public void Add(Exception exception);
    public void Add(string errorMessage);
}

[Serializable]
public class ModelError
{
    public ModelError(Exception exception);
    public ModelError(string errorMessage);
    public ModelError(Exception exception, string errorMessage);

    public string ErrorMessage { get; private set; }
    public Exception Exception { get; private set; }
}
```

实例演示：Model 绑定过程中对 ModelState 的设置（S509）

在 `Model` 绑定过程中，`ModelBinder` 除了为目标 `Action` 方法的执行提供参数列表之外，还会对基于当前 `Controller` 的 `ModelState` 进行设置，现在通过一个简单的实例来证明这一点。我们在一个 `ASP.NET MVC` 应用中定义了如下一个 `HomeController`，在基于 `HTTP-GET` 的 `Action` 方法 `Index` 中，创建了一个 `Contact` 对象并作为 `Model` 呈现在默认的 `View` 中。另一个基于应用了 `HttpPostAttribute` 特性的 `Index` 方法具有一个 `Contact` 类型的参数，该方法将当前 `ModelState` 作为 `Model` 呈现在一个名为“`ModelState`”的 `View` 中。

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        Address address = new Address
        {
            Province    = "江苏",
            City        = "苏州",
            District    = "工业园区",
            Street      = "星湖街 328 号"
        };
        Contact contact = new Contact
        {
            Name        = "张三",
            PhoneNo     = "123456789",
            EmailAddress = "zhangsan@gmail.com",
            Address      = address
        };
        return View(contact);
    }

    [HttpPost]
    public ActionResult Index(Contact contact)
    {
        return View("ModelState", this.ViewData.ModelState);
    }
}

public class Contact
{
    public string Name { get; set; }
    public string PhoneNo { get; set; }
    public string EmailAddress { get; set; }
    public Address Address { get; set; }
}

public class Address
{
    public string Province { get; set; }
    public string City { get; set; }
    public string District { get; set; }
    public string Street { get; set; }
}

```

下面是 Action 方法 Index 默认 View 的定义，可以看出这是一个 Model 类型为 Contact 的强类型 View。在该 View 中我们将作为 Model 的 Contact 对象以编辑模式呈现在一个表单中，并在表单中添加提交按钮。有的读者可能会觉得奇怪：为什么在调用 `Html<TModel>` 的模板方法 `EditorForModel` 对 Model 对象进行呈现之外，还需要调用 `EditorFor` 方法将 Contact 对象的 Address 属性进行基于编辑模式的呈现呢？原因在于 Contact 的 Address 属性是复杂类型，默认情况下针对 Contact 的 `EditorForModel` 方法输出的 HTML 并不会包括其 Address 属性的内容，相关的内容在第 4 章“Model 元数据的解析”中具有详细的介绍。

```

@model Contact
<html>
    <head>
        <title>修改 Contact 信息</title>
    </head>

```

```

<body>
    @using (Html.BeginForm())
    {
        @Html.EditorForModel()
        @Html.EditorFor(m=>m.Address)
        <input type="submit" value="保存" />
    }
</body>
</html>

```

另一个用于呈现当前 ModelState 的 View 具有如下的定义, 这是一个 Model 类型为 ModelStateDictionary 的强类型 View, 在该 View 中我们将作为 Model 的 ModelStateDictionary 对象的 Key 和 Value 以表格的形式呈现出来。

```

@model ModelStateDictionary
<html>
<head>
    <title>ModelState</title>
</head>
<body>
    <table>
        <tr><th>Key</th><th>Value</th></tr>
        @foreach (var item in Model)
        {
            <tr>
                <td>@item.Key</td>
                <td>@item.Value.Value.ConvertTo(typeof(string))</td>
            </tr>
        }
    </table>
</body>
</html>

```

运行该程序之后, 一个用于编辑联系人信息的页面会被呈现出来。直接点击“保存”按钮提交表单, 用于输出当前 ModelState 结构的页面会随之出现。如图 5-12 所示, ModelState 中的值与提交的表单具有相同的结构和数据值。

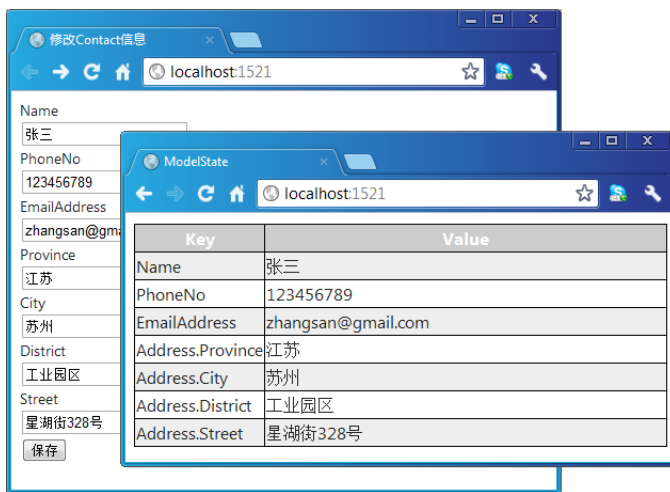


图 5-12 ModelBinder 针对 ModelState 的设置

5.3.3 ModelBindingContext 的创建

通过前面的介绍我们知道最终的 Model 绑定是通过调用相应 ModelBinder 的 BindModel 方法来完成的, 该方法具有两个基于上下文的参数, 一个是代表当前 Controller 上下文的 ControllerContext 对象, 另一个则是针对当前 Model 绑定上下文的 ModelBindingContext 对象。前者在 Controller 激活的时候就已经创建了, 那么只要我们能够针对当前 Model 绑定创建出相应的 ModelBindingContext 对象, 整个 Model 绑定就迎刃而解。

在正式介绍 ModelBindingContext 的创建过程之前先来看看 ModelBindingContext 的定义。如下面的代码片段所示, ModelBindingContext 具有一系列属性。由于 Model 绑定是针对描述 Action 方法的某个参数的元数据进行的, 所以这些属性基本上来源于用于描述 Action 方法参数的 ParameterDescriptor 对象。

```
public class ModelBindingContext
{
    public string          ModelName { get; set; }
    public Type            ModelType { get; set; }
    public ModelMetadata    ModelMetadata { get; set; }
    public IDictionary<string, ModelMetadata> PropertyMetadata { get; }

    public object          Model { get; set; }
    public ModelStateDictionary ModelState { get; set; }
    public IValueProvider   ValueProvider { get; set; }

    public bool            FallbackToEmptyPrefix { get; set; }
    public Predicate<string> PropertyFilter { get; set; }
}
```

整个 Model 绑定在 Action 的执行过程中进行, 而 Action 的执行是通过组件 ActionInvoker 来完成的。当 ActionInvoker 在执行目标 Action 的时候, 会得到用于描述 Action 的 ActionDescriptor 对象。然后它遍历 ActionDescriptor 的参数列表, 并根据对应的 ParameterDescriptor 对象结合当前 Controller 上下文创建 ModelBindingContext。最后通过上面介绍的 ModelBinder 提供机制获取对应的 ModelBinder, 被创建的 ModelBindingContext 对象作为参数调用 ModelBinder 的 GetModel 方法得到的对象就是相应的参数值。最终数据值的提供是通过 ValueProvider 完成的, 它提供的数据值同时以 ModelState 的形式保存起来。

ModelBindingContext 的 ModelName 和 ModelType 分别表示 Model 的名称和类型, 而 Model 的类型自然就是参数类型。如果通过 ParameterDescriptor 的 Prefix 属性表示的前缀不为空, 那么该前缀将会作为 Model 的名称, 否则 Model 的名称就是参数名。ValueProvider 在进行数据提供过程中就是通过 ModelName 属性进行数据匹配的。

ModelBindingContext 的 ModelMetadata 表示针对参数类型的 Model 元数据, 而参数类型属性列表的 Model 元数据被保存在 PropertyMetadata 属性中。PropertyMetadata 属性类型为 IDictionary<string, ModelMetadata>, 它的 Key 表示属性名称。Model 元数据信息通过注册的 ModelMetadataProvider 来提供, 具体的 Model 元数据提供机制请参见第 4 章“Model 元数据

的解析”。

ModelBindingContext 的 Model 表示最终绑定的参数值，其 ModelState 属性和 Controller 的 ViewData 的同名属性引用同一个 ModelStateDictionary 对象，所以 Model 绑定过程中对 ModelState 的设置会反映在 Controller 的 ViewData 上，用于提供具体数据值的 ValueProvider 来源于 Controller 的同名属性。

如果没有利用 BindAttribute 特性为参数设置一个前缀，默认情况下会将参数名称作为前缀。通过前面的介绍我们知道，这个前缀会被 ValueProvider 用于数据的匹配。如果 ValueProvider 通过此前缀找不到匹配的数据，将剔除前缀再次进行数据获取。针对如下定义的 Action 方法 AddContacts，在请求数据并不包含基于参数名（“foo”和“bar”）前缀的情况下，两个参数最终将被绑定上相同的值。

```
public class ContactController
{
    public void AddContacts(Contact foo, Contact bar)
    {
        //省略实现
    }
}
```

反之，如果我们应用 BindAttribute 特性显式地设置了一个前缀，这种去除前缀再次实施 Model 绑定的后备机制将不会被采用，是否采用后备 Model 绑定策略通过 ModelBindingContext 的 FallbackToEmptyPrefix 属性来控制。

最后一个参数 PropertyFilter 返回的是一个 Predicate<string>类型的委托，布尔类型的返回值表示指定的属性是否会参与 Model 的绑定。具体的逻辑依赖于 ParameterDescriptor 的 BindingInfo 属性表示的 ParameterBindingInfo 对象。具体来说，如果该对象的 Include 属性（参与绑定属性集合）为空或者不包含指定的属性，并且指定的属性不包含在 Exclude 属性（不参与绑定的属性集合）中，该委托对象返回 True，否则返回 False。

由于数据值的提供最终通过 ValueProvider 来完成，而它只能提供简单类型的数据，所以对于简单类型参数的 Model 绑定可以直接通过 ValueProvider 来提供。但是对于复杂类型参数来说，则会遍历属性列表，按照相同的方式提供其对应的属性值，所以 Model 绑定是一个递归的过程。那么默认情况下采用的 Model 绑定具有怎样的流程，我们将在下面一节对其进行详细地讲述。

5.4 Model 绑定的默认实现

总的来说，针对目标 Action 方法参数的 Model 绑定完全由组件 ModelBinder 来完成，在默认情况下使用的 ModelBinder 类型为 System.Web.Mvc.DefaultModelBinder。接下来我们以实例演示的方式逐层深入地讲述 ASP.NET MVC 实现在 DefaultModelBinder 中的默认 Model 绑定机制。

5.4.1 简单类型

对于旨在绑定目标 Action 方法参数值的 Model 来说，最简单的莫过于简单参数类型的绑定。通过第 4 章“Model 元数据的提供”的介绍我们知道复杂类型和简单类型之间的区别仅仅在于它们是否支持针对字符串类型的转换。参数对象的数据源均以字符串形式存在于请求之中（比如提交的表单、JSON 字符串以及查询字符串等），而 ValueProvider 也只能实现基于简单类型的数据提供。对于基于简单类型参数的 Model 绑定，只需要将参数名称或者通过应用在参数上的 BindAttribute 特性指定的前缀作为 Key 调用 ValueProvider 的 GetValue 方法，并将得到的 ValueProviderResult 对象转换成参数类型即可。

为了演示针对简单数据类型的 Model 绑定，我们自定义了如下一个 DefaultModelBinder，它是对 ASP.NET MVC 默认使用的 DefaultModelBinder 的模拟。随着 Model 绑定类型的复杂化，我们将对它进行逐渐地完善，最终的 DefaultModelBinder 将基本能够体现真正的 Model 绑定的实现。

```
public class DefaultModelBinder: IModelBinder
{
    public object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        return this.GetModel(controllerContext, bindingContext.ModelType,
            bindingContext.ValueProvider, bindingContext.ModelName);
    }

    public object GetModel(ControllerContext controllerContext, Type modelType,
        IValueProvider valueProvider, string key)
    {
        if (!valueProvider.ContainsPrefix(key))
        {
            return null;
        }
        return valueProvider.GetValue(key).ConvertTo(modelType);
    }
}
```

实现在 DefaultModelBinder 的 Model 绑定体现在 GetModel 方法中，三个参数分别表示 Model 类型、ValueProvider 对象和被它用于获取数据的 Key。对于简单类型参数的 Model 绑定来说，Model 类型为参数类型，而 Key 实际上是参数名称或者是参数上应用的 BindAttribute 特性指定的前缀。这三个对象都可以通过表示 Model 绑定上下文的 ModelBindingContext 对象获取。具体参数值最终通过调用 ValueProvider 的 GetValue 来获取，该方法返回的是一个 ValueProviderResult 对象，我们需要调用它的 ConvertTo 方法转换成参数类型。

现在我们通过一个具体的实例来演示 ASP.NET MVC 在执行目标 Action 的时候是如何基于参数创建 Model 绑定上下文，并最终利用 ModelBinder 来提供具体参数值的。在一个 ASP.NET MVC 应用中定义了如下一个 DemoController。

```

public abstract class DemoController : Controller
{
    public IModelBinder ModelBinder { get; private set; }

    public DemoController()
    {
        this.ModelBinder = new DefaultModelBinder();
        this.ValueProvider = this.CreateValueProvider();
    }

    protected abstract IValueProvider CreateValueProvider();

    protected ActionResult InvokeAction(string actionName)
    {
        ControllerDescriptor controllerDescriptor =
            new ReflectedControllerDescriptor(this.GetType());
        ActionDescriptor actionDescriptor = controllerDescriptor.FindAction(
            ControllerContext, actionName);
        Dictionary<string, object> parameters = new Dictionary<string, object>();
        foreach (ParameterDescriptor parameterDescriptor
            in actionDescriptor.GetParameters())
        {
            string modelName = parameterDescriptor.BindingInfo.Prefix
                ?? parameterDescriptor.ParameterName;

            ModelBindingContext bindingContext = new ModelBindingContext
            {
                FallbackToEmptyPrefix =
                    parameterDescriptor.BindingInfo.Prefix == null,
                ModelMetadata =
                    ModelMetadataProviders.Current.GetMetadataForType(null,
                        parameterDescriptor.ParameterType),
                ModelName = modelName,
                ModelState = ModelState,
                ValueProvider = this.ValueProvider
            };
            parameters.Add(parameterDescriptor.ParameterName,
                this.ModelBinder.BindModel(ControllerContext, bindingContext));
        }
        return (ActionResult)actionDescriptor.Execute(ControllerContext,
            parameters);
    }
}

```

我们在 `DemoController` 中定义了一个用于完成 Model 绑定的 `ModelBinder` 属性，在构造函数中该属性被初始化成我们自定义的 `DefaultModelBinder`，另一个抽象方法 `CreateValueProvider` 用于创建提供数据的 `ValueProvider` 对象。

核心 `InvokeAction` 方法用于执行指定的 Action（假定 Action 方法返回一个 `ActionResult` 对象）。在该方法中我们针对自身类型创建出相应的 `ControllerDescriptor` 对象，并根据 Action 名称得到对应的 `ActionDescriptor` 对象。然后根据 `ActionDescriptor` 获取用于描述其参数的 `ParameterDescriptor` 列表，而 Model 绑定就是根据每个 `ParameterDescriptor` 进行的。

具体来说，根据针对 `ParameterDescriptor` 对象创建表示 Model 绑定上下文的 `ModelBindingContext` 对象，并对其 `ModelName`、`ModelState`、`ValueProvider`、`ModelMetadata`

和 `FallbackToEmptyPrefix` 属性进行了初始化。最后将该 `ModelBindingContext` 对象和当前 `ControllerContext` 作为参数调用 `ModelBinder` 的 `BindModel` 方法得到相应的参数值。所有的参数最终被添加到一个字典对象中(Key 表示参数名称), 它会作为参数调用 `ActionDescriptor` 的 `Execute` 方法实现对指定 `Action` 的执行。

我们接下来创建如下一个继承自 `DemoController` 的 `HomeController`。`Action` 方法 `DemoAction` 具有三个简单类型的参数 `foo`、`bar` 和 `baz`, 其中最后一个参数上应用了 `BindAttribute` 特性将绑定前缀指定为“`qux`”。我们将三个参数值添加到一个字典对象中(Key 表示参数名称), 并将其作为 `Model` 对象呈现在对应的 `View` 中。

```
public class HomeController : DemoController
{
    protected override IValueProvider CreateValueProvider()
    {
        NameValueCollection dataSource = new NameValueCollection();
        dataSource.Add("Foo", "ABC");
        dataSource.Add("Bar", "123");
        dataSource.Add("Baz", "456.01");
        dataSource.Add("Qux", "789.01");
        return new NameValueCollectionValueProvider(dataSource,
            CultureInfo.CurrentCulture);
    }

    public ActionResult Index()
    {
        return this.InvokeAction("DemoAction");
    }

    public ActionResult DemoAction(
        string foo,
        int bar,
        [Bind(Prefix="qux")]
        double baz)
    {
        Dictionary<string, object> parameters = new Dictionary<string, object>();
        parameters.Add("foo", foo);
        parameters.Add("bar", bar);
        parameters.Add("baz", baz);
        return View("DemoAction", parameters);
    }
}
```

实现的抽象方法 `CreateValueProvider` 返回一个 `NameValueCollectionValueProvider` 对象, 针对 `DemoAction` 方法的参数定义, 作为数据容器的 `NameValueCollection` 包含了对应的数据项 (`foo`、`bar`、`baz` 和 `qux`)。在默认的动作方法 `Index` 中, 我们调用 `InvokeAction` 方法实现对 `Action` 方法 `DemoAction` 的执行。

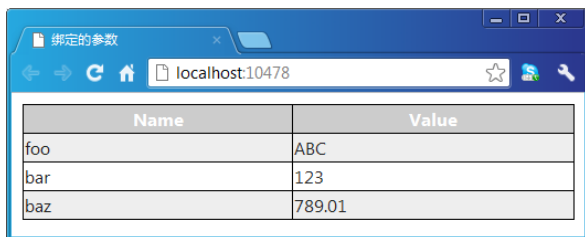
如下所示的是 `Action` 方法 `DemoAction` 对应 `View` 的定义, 可以看出这是一个 `Model` 类型为 `IDictionary<string, object>` 的强类型 `View`。在该 `View` 中, 作为 `Model` 的字典的 `Key` 和 `Value` 通过一个表格呈现出来。

```

@model IDictionary<string, object>
<html>
  <head>
    <title>绑定的参数</title>
  </head>
  <body>
    <table>
      <tr><th>Name</th><th>Value</th></tr>
      @foreach (var item in Model)
      {
        <tr><td>@item.Key</td><td>@item.Value</td></tr>
      }
    </table>
  </body>
</html>

```

该程序运行之后将在浏览器中得到如图 5-13 所示的输出结果，可以看出 Action 方法 DemoAction 得到正常的执行，而呈现出来的就是绑定的参数列表。作为 ValueProvider 数据容器的 NameValueCollection 中包含 Key 分别为“baz”和“qux”的两个数据项，由于 DemoAction 的参数 baz 上应用了 BindAttribute 特性将绑定前缀设置为“qux”，所以对应的 Key 是“qux”而不是参数名“baz”。(S510)



Name	Value
foo	ABC
bar	123
baz	789.01

图 5-13 简单类型的 Model 绑定

5.4.2 复杂类型

对于简单类型的参数来说，由于支持与字符串类型之间的转换，相应 ValueProvider 可以从数据源中提取相应的数据并直接转换成参数类型，所以针对简单类型的 Model 绑定是一步到位的过程，但是针对复杂类型的 Model 绑定就没有这么简单了。

复杂对象可以表示为一个树形层次化结构，其对象本身和属性代表相应的节点，叶子节点代表简单数据类型属性。ValueProvider 采用的数据容器具有一个扁平的数据结构，它通过采用“属性名称链”为前缀的 Key 实现与这个“对象树”中的叶子节点的映射。

```

public class Contact
{
    public string Name { get; set; }
    public string PhoneNo { get; set; }
    public string EmailAddress { get; set; }
    public Address Address { get; set; }
}

```

```
public class Address
{
    public string Province { get; set; }
    public string City { get; set; }
    public string District { get; set; }
    public string Street { get; set; }
}
```

以我们熟悉的 `Contact` 类型为例，它具有三个简单类型的属性（`Name`、`PhoneNo` 和 `EmailAddress`）和复杂类型 `Address` 的属性，而一个 `Address` 对象又具有四个简单类型的属性。一个 `Contact` 对象的数据结构可以通过如图 5-14 所示的树来表示，这棵树的所有叶子节点均为简单类型的属性。如果需要通过一个 `ValueProvider` 来构建一个完整的 `Contact` 对象，它必须能够提供所有叶子节点的数值，而 `ValueProvider` 通过基于属性名称前缀的 `Key` 实现与对应的叶子节点的映射。

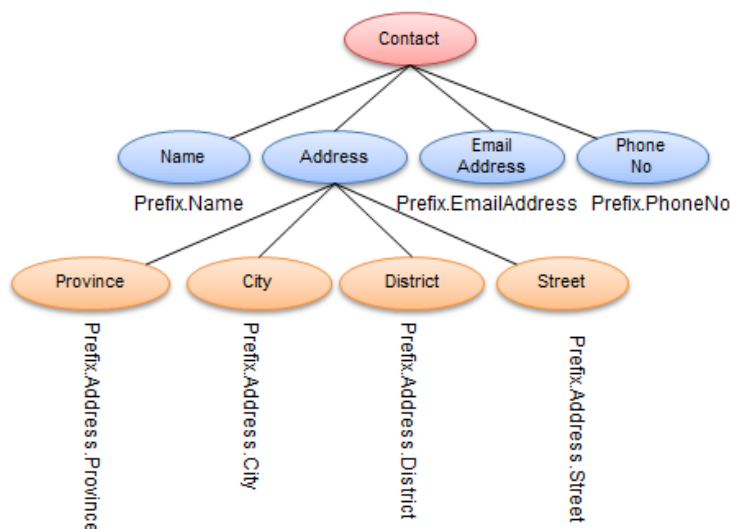


图 5-14 复杂对象层次化结构中叶子节点对应的 Key

实际上当我们调用 `HtmlHelper<TModel>` 的模板方法 `EditorFor/EditorForModel` 的时候就是按照这样的匹配方式对表单元素进行命名的。假设在一个将 `Contact` 作为 `Model` 类型的强类型 `View` 中，我们按照如下的方式调用 `HtmlHelper<TModel>` 的扩展方法 `EditorFor` 将 `Model` 对象的所有信息以编辑的模式呈现出来。

```
@model Contact
@Html.EditorFor(m => m.Name)
@Html.EditorFor(m => m.PhoneNo)
@Html.EditorFor(m => m.EmailAddress)
@Html.EditorFor(m => m.Address.Province)
@Html.EditorFor(m => m.Address.City)
@Html.EditorFor(m => m.Address.District)
@Html.EditorFor(m => m.Address.Street)
```

下面的代码片段代表了作为 Model 对象的 Contact 在最终呈现出来的 View 中的 HTML，可以清楚地看到这些<input>表单元素完全是根据属性名称和类型层次结构进行命名的。顺便提一下，对于基于提交表单的 Model 绑定来说，作为匹配的是表单元素的 name 属性而非 id 属性，所以这里的命名指的是 name 属性而非 id 属性。

```
<input id="Name" name="Name" type="text" ... />
<input id="PhoneNo" name="PhoneNo" type="text" ... />
<input id="EmailAddress" name="EmailAddress" type="text" ... />
<input id="Address_Province" name="Address.Province" type="text" ... />
<input id="Address_City" name="Address.City" type="text" ... />
<input id="Address_District" name="Address.District" type="text" ... />
<input id="Address_Street" name="Address.Street" type="text"... />
```

针对复杂类型的 Model 绑定是一个递归的过程，它先通过反射根据数据类型创建相应的对象，然后绑定其属性值。每一次递归都会将属性名称附加到现有前缀上作为下一级递归的前缀。现在我们将针对复杂类型的 Model 绑定实现在自定义的 DefaultModelBinder 之上。

首先在 DefaultModelBinder 中定义如下一个根据指定类型创建相应对象的 CreateModel 方法，默认通过调用 Activator 的 CreateInstance 方法创建给定类型的对象。如果指定的类型是 IDictionary<> 接口，我们会创建一个 Dictionary<> 对象；如果是 IEnumerable<>、ICollection<> 或者 IList<> 接口，则会创建一个 List<> 对象。

```
public class DefaultModelBinder : IModelBinder
{
    //其他成员
    private object CreateModel(Type modelType)
    {
        Type type = modelType;
        if (modelType.IsGenericType)
        {
            Type genericTypeDefinition = modelType.GetGenericTypeDefinition();
            if (genericTypeDefinition == typeof(IDictionary<>))
            {
                type = typeof(Dictionary<>).MakeGenericType(
                    modelType.GetGenericArguments());
            }
            else if (((genericTypeDefinition == typeof(IEnumerable<>))
                || (genericTypeDefinition == typeof(ICollection<>)))
                || (genericTypeDefinition == typeof(IList<>)))
            {
                type = typeof(List<>).MakeGenericType(
                    modelType.GetGenericArguments());
            }
        }
        return Activator.CreateInstance(type);
    }
}
```

然后我们定义了如下一个 GetComplexModel 方法，根据指定的数据类型、ValueProvider 和前缀来创建一个对应的数据对象。在该方法中，我们先调用上面定义的 CreateModel 方法根据指定的类型创建一个相应的对象，然后遍历复杂类型的属性列表，将提供的前缀和属性

名组成一个新的 Key，并以此 Key 调用 GetModel 方法得到对应的属性值，最终通过反射对属性进行赋值。

```
public class DefaultModelBinder : IModelBinder
{
    //其他成员
    protected virtual object GetComplexModel(ControllerContext controllerContext,
        Type modelType, IValueProvider valueProvider, string prefix)
    {
        object model = CreateModel(modelType);
        foreach (PropertyDescriptor property in
            TypeDescriptor.GetProperties(modelType))
        {
            if (property.IsReadOnly)
            {
                continue;
            }
            string key = string.IsNullOrEmpty(prefix) ? property.Name : prefix +
                "." + property.Name;
            property.SetValue(model, GetModel(controllerContext,
                property.PropertyType, valueProvider, key));
        }
        return model;
    }
}
```

我们将针对 GetComplexModel 方法调用添加到 GetModel 方法中。如下面的代码片段所示，我们根据数据类型创建出用于描述 Model 元数据的 ModelMetadata 对象，根据它的 IsComplexType 属性判断参数类型是否是复杂类型，如果是则调用 GetComplexModel 方法来获取对应的复杂对象。

```
public class DefaultModelBinder : IModelBinder
{
    public object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        object model = this.GetModel(controllerContext, bindingContext.ModelType,
            bindingContext.ValueProvider, bindingContext.ModelName);
        if (bindingContext.FallbackToEmptyPrefix && null == model)
        {
            model = this.GetModel(controllerContext, bindingContext.ModelType,
                bindingContext.ValueProvider, "");
        }
        return model;
    }

    public virtual object GetModel(ControllerContext controllerContext,
        Type modelType, IValueProvider valueProvider, string key)
    {
        if (!valueProvider.ContainsPrefix(key))
        {
            return null;
        }

        ModelMetadata modelMetadata = ModelMetadataProviders.Current
            .GetMetadataForType(null, modelType);
```

```

        if (!modelMetadata.IsComplexType)
        {
            return valueProvider.GetValue(key).ConvertTo(modelType);
        }
        if (modelMetadata.IsComplexType)
        {
            return GetComplexModel(controllerContext, modelType, valueProvider,
                                   key);
        }
        return null;
    }
}

```

前面我们提到在进行 Model 绑定的时候,如果没有在参数上应用 `BindAttribute` 特性设置一个绑定前缀, `DefaultModelBinder` 会默认采用参数名称作为前缀。如果根据这个前缀获取不到相应的数据,它会在剔除该前缀的情况下再进行一次 Model 绑定。`ModelBindingContext` 类型的 `FallbackToEmptyPrefix` 属性表示是否需要采用空字符串作为前缀再次实施 Model 绑定,这样的机制也体现在前面定义的 `BindModel` 方法中。

我们使用前面用于演示简单类型 Model 绑定的实例来演示针对复杂类型的 Model 绑定,为此我们对 `HomeController` 的 `DemoAction` 进行了重新定义,让它包含两个 `Contact` 类型的参数(`foo` 和 `bar`),并将它们相关的属性添加到一个字典对象中作为 Model 呈现在默认的 View 中。实现的 `CreateValueProvider` 方法中返回一个 `NameValueCollectionValueProvider`,作为数据容器的 `NameValueCollection` 提供了一个 `Contact` 对象的所有属性数据。

```

public class HomeController : DemoController
{
    //其他成员
    protected override IValueProvider CreateValueProvider()
    {
        NameValueCollection dataSource = new NameValueCollection();

        dataSource.Add("Name", "张三");
        dataSource.Add("PhoneNo", "123456789");
        dataSource.Add("EmailAddress", "zhangsan@gmail.com");

        dataSource.Add("Address.Province", "江苏");
        dataSource.Add("Address.City", "苏州");
        dataSource.Add("Address.District", "工业园区");
        dataSource.Add("Address.Street", "星湖街 328 号");
        return new NameValueCollectionValueProvider(dataSource,
            CultureInfo.CurrentCulture);
    }

    public ActionResult DemoAction(Contact foo, Contact bar)
    {
        Dictionary<string, object> parameters = new Dictionary<string, object>();

        parameters.Add("foo.Name", foo.Name);
        parameters.Add("foo.PhoneNo", foo.PhoneNo);
        parameters.Add("foo.EmailAddress", foo.EmailAddress);
        Address address = foo.Address;
    }
}

```

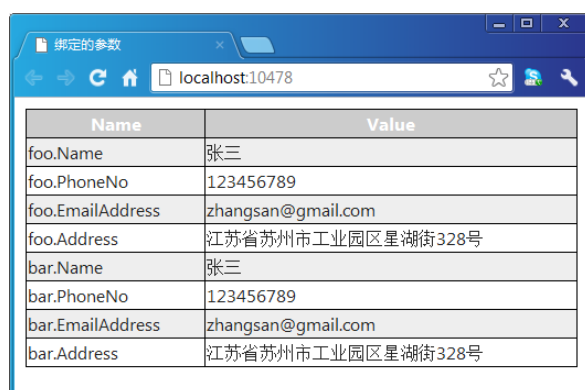
```

parameters.Add("foo.Address", string.Format("{0}省{1}市{2}{3}",
    address.Province, address.City, address.District, address.Street));

parameters.Add("bar.Name", bar.Name);
parameters.Add("bar.PhoneNo", bar.PhoneNo);
parameters.Add("bar.EmailAddress", bar.EmailAddress);
address = bar.Address;
parameters.Add("bar.Address", string.Format("{0}省{1}市{2}{3}",
    address.Province, address.City, address.District, address.Street));
return View("DemoAction", parameters);
}
}

```

运行该程序后会在浏览器中呈现出如图 5-15 所示的输出结果，可以看出两个 DemoAction 方法的两个参数被绑定上了相同的数据。(S511)



Name	Value
foo.Name	张三
foo.PhoneNo	123456789
foo.EmailAddress	zhangsan@gmail.com
foo.Address	江苏省苏州市工业园区星湖街328号
bar.Name	张三
bar.PhoneNo	123456789
bar.EmailAddress	zhangsan@gmail.com
bar.Address	江苏省苏州市工业园区星湖街328号

图 5-15 复杂类型的 Model 绑定 (1)

之所以同一个 Action 方法中两个相同类型的参数会绑定相同的数据，就是源于上面介绍的“去除前缀的后备 Model 绑定机制”。现在我们对 HomeController 的 CreateValueProvider 方法进行如下修改使 ValueProvider 的数据容器中包含两组 Contact 数组，对应的 Key 采用了 Action 方法的参数名称作为前缀。

```

public class HomeController : DemoController
{
    //其他成员
    protected override IValueProvider CreateValueProvider()
    {
        NameValueCollection dataSource = new NameValueCollection();

        dataSource.Add("foo.Name", "张三");
        dataSource.Add("foo.PhoneNo", "123456789");
        dataSource.Add("foo.EmailAddress", "zhangsan@gmail.com");
        dataSource.Add("foo.Address.Province", "江苏");
        dataSource.Add("foo.Address.City", "苏州");
        dataSource.Add("foo.Address.District", "工业园区");
        dataSource.Add("foo.Address.Street", "星湖街 328 号");
    }
}

```



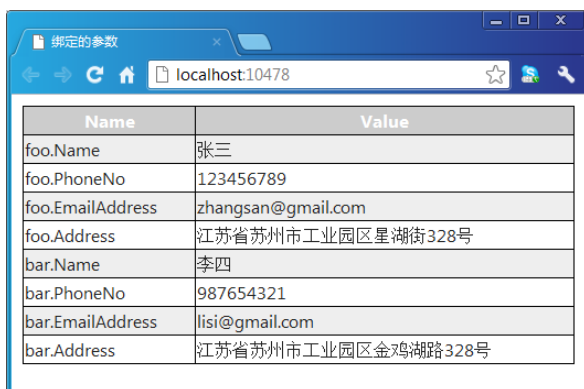
```

        dataSource.Add("bar.Name", "李四");
        dataSource.Add("bar.PhoneNo", "987654321");
        dataSource.Add("bar.EmailAddress", "lisi@gmail.com");
        dataSource.Add("bar.Address.Province", "江苏");
        dataSource.Add("bar.Address.City", "苏州");
        dataSource.Add("bar.Address.District", "工业园区");
        dataSource.Add("bar.Address.Street", "金鸡湖路 328 号");

        return new NameValueCollectionValueProvider(dataSource,
            CultureInfo.CurrentCulture);
    }
}

```

再次运行我们的程序后将会发现 DemoAction 方法的两个 Contact 类型的参数分别绑定了不同的数据，具体的运行结果如图 5-16 所示。（S512）



Name	Value
foo.Name	张三
foo.PhoneNo	123456789
foo.EmailAddress	zhangsan@gmail.com
foo.Address	江苏省苏州市工业园区星湖街328号
bar.Name	李四
bar.PhoneNo	987654321
bar.EmailAddress	lisi@gmail.com
bar.Address	江苏省苏州市工业园区金鸡湖路328号

图 5-16 复杂类型的 Model 绑定（2）

5.4.3 数组

基于数组和集合类型的 Model 绑定机制比较类似。作为 NameValueCollection ValueProvider 数据容器的 NameValueCollection 对象并没有对 Key 作唯一性约束，如果目标对象是一个简单类型的数组或者集合，同一个 Key 对应的多个 Value 将会转换成目标数组或者集合的元素。此外，针对数组/集合的 Model 绑定还支持索引。

基于同名数据项的数组绑定

对于 NameValueCollectionProvider 来说，其 GetValue 方法得到的 ValueProviderResult 的 RawValue 属性总是一个字符串数组。当我们调用 ValueProviderResult 的 ConvertTo 方法将提供的值转换成某种类型时，如果目标类型为一个简单类型，那么第一个字符串将会提取出来并转换为目标类型。如果目标类型是一个元素为简单类型的数组或者集合，每个字符串都会被提取出来并转换为目标数组/集合对象的元素。

NameValueCollectionProvider（包括 FormValueProvider 和 QueryStringValueProvider）的数据提供机制决定了 Model 绑定的默认行为。如果绑定的目标对象是一个数组/集合，匹配的同名数据项将会作为目标对象的元素。实际上 HttpFileCollectionValueProvider 也采用了类似的机制，即如果绑定的目标对象类型是一个 HttpPostedFileBase 数组，那么匹配的同名文件输入元素都将作为其数据源。

```
<input name="Foo" type="text" ... />
<input name="Foo" type="text" ... />
<input name="Foo" type="text" ... />

<input name="Bar" type="file" .../>
<input name="Bar" type="file" .../>
<input name="Bar" type="file" .../>
```

假设我们在某个 Controller 中定义了如下一个 Action 方法 DemoAction，它具有两个数组类型的参数 foo 和 bar，元素类型分别是字符串和 HttpPostedFileBase。如果针对该 Action 的请求具有包含如上<input>元素的表单，那么两组<input>元素的值将会分别绑定到这两个名称匹配的参数上。

```
Public void DemoAction(string[] foo, HttpPostedFileBase[] bar)
```

现在我们将针对同名数据项的数组绑定实现在我们自定义的 DefaultModelBinder 中。如下面的代码片段所示，定义了一个 GetArrayModel 来获取目标类型为数组的对象，在该方法中，我们直接将传入的前缀作为参数调用 ValueProvider 的 GetValue 方法，并将得到的 ValueProviderResult 转换成目标数组类型。在 GetModel 方法中，如果传入的数据类型为数组，则调用 GetArrayModel 方法返回绑定的对象。

```
public class DefaultModelBinder : IModelBinder
{
    //其他成员
    public virtual object GetModel(ControllerContext controllerContext,
        Type modelType, IValueProvider valueProvider, string key)
    {
        if (!valueProvider.ContainsPrefix(key))
        {
            return null;
        }

        if (modelType.IsArray)
        {
            return GetArrayModel(controllerContext, modelType, valueProvider,
                key);
        }
        //其他操作
    }

    protected virtual object GetArrayModel(
        ControllerContext controllerContext, Type modelType,
        IValueProvider valueProvider, string prefix)
```

```

    {
        if (valueProvider.ContainsPrefix(prefix)
            && !string.IsNullOrEmpty(prefix))
        {
            ValueProviderResult result = valueProvider.GetValue(prefix);
            if (null != result)
            {
                return result.ConvertTo(modelType);
            }
        }
        return null;
    }
}

```

为了演示针对同名数据项的数组绑定,我们对前面演示实例中的 `HomeController` 进行了如下的修改: `Action` 方法 `DemoAction` 具有两个数组类型的参数 `foo` 和 `bar`,我们将两个数组的元素封装到一个字典对象中并作为 `Model` 呈现在默认的 `View` 中。对于通过 `CreateValueProvider` 方法创建的 `NameValueCollectionValueProvider`,作为其数据容器的 `NameValueCollection` 中具有两组数据,其 `Key` 分别是 `DemoAction` 方法的参数名。

```

public class HomeController : DemoController
{
    protected override IValueProvider CreateValueProvider()
    {
        NameValueCollection dataSource = new NameValueCollection();

        dataSource.Add("foo", "abc");
        dataSource.Add("foo", "ijk");
        dataSource.Add("foo", "xyz");

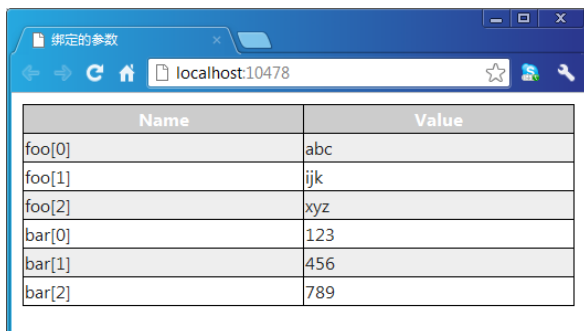
        dataSource.Add("bar", "123");
        dataSource.Add("bar", "456");
        dataSource.Add("bar", "789");

        return new NameValueCollectionValueProvider(dataSource,
            CultureInfo.CurrentCulture);
    }

    public ActionResult DemoAction(string[] foo, int[] bar)
    {
        Dictionary<string, object> parameters = new Dictionary<string, object>();
        for (int i = 0; i < foo.Length; i++)
        {
            parameters.Add(string.Format("foo[{0}]", i), foo[i]);
        }
        for (int i = 0; i < bar.Length; i++)
        {
            parameters.Add(string.Format("bar[{0}]", i), bar[i]);
        }
        return View("DemoAction", parameters);
    }
}

```

程序运行后会在浏览器中呈现出如图 5-17 所示的输出结果,可以看出 `Action` 方法 `DemoAction` 的两个参数得到了正常的绑定。(S513)



Name	Value
foo[0]	abc
foo[1]	ijk
foo[2]	xyz
bar[0]	123
bar[1]	456
bar[2]	789

图 5-17 数组类型的 Model 绑定 (1)

基于索引的数组绑定

ValueProvider 基于索引的匹配策略可以通过 `HtmlHelper<TModel>` 的模板方法 `EditorFor` 来体现。如下面的代码片段所示，在一个 Model 为 `Contact` 数组的强类型 View 中，我们调用它的扩展方法 `EditorFor` 将数组的前两个元素的属性以编辑模式呈现出来。

```
@model Contact[]
@Html.EditorFor(m => m[0].Name)
@Html.EditorFor(m => m[0].PhoneNo)
@Html.EditorFor(m => m[0].EmailAddress)

@Html.EditorFor(m => m[1].Name)
@Html.EditorFor(m => m[1].PhoneNo)
@Html.EditorFor(m => m[1].EmailAddress)
```

上面这段代码最终会生成如下一段 HTML，6 次 `EditorFor` 方法的调用转换为 6 个文本框，我们可以清楚地看到它们的名称被添加了 “[0]” 和 “[1]” 这样的索引前缀。如果这些元素存在于一个提交的表单中，并且目标 Action 方法包含一个匹配的 `Contact` 数组类型的参数，选用的 `ModelBinder` 会为之生成一个包含两个元素的 `Contact` 数组作为其参数。

```
<input name="[0].Name" type="text" value="" .../>
<input name="[0].PhoneNo" type="text" value="" .../>
<input name="[0].EmailAddress" type="text" value="" .../>

<input name="[1].Name" type="text" value="" .../>
<input name="[1].PhoneNo" type="text" value="" .../>
<input name="[1].EmailAddress" type="text" value="" .../>
```

基于数组的 Model 绑定采用“零基索引”，同时要求索引在数值上必须是连续的。举个简单的例子，假设提交的表单中具有如下 6 个采用索引方式命名的 `hidden` 元素，但是索引数值并不是连续的（缺了一个 [3]）。

```
<input name="[0]" type="hidden" value="foo" />
<input name="[1]" type="hidden" value="bar" />
<input name="[2]" type="hidden" value="baz" />

<input name="[4]" type="hidden" value="123" />
```

```
<input name="[5]" type="hidden" value="456" />
<input name="[6]" type="hidden" value="789" />
```

如果包含此表单的请求针对如下一个具有一个字符串数组参数的 **Action** 方法，上述的 `<input>` 元素值将会绑定到参数 `array` 上。但是由于索引值不具有连续性，会导致后面的三个 `<input>` 元素值（“123”、“456”和“789”）被丢弃，也就是说绑定后的 `array` 参数值仅仅具有三个元素（“foo”、“bar”和“baz”）。

```
public ActionResult Index(string[] array);
```

除了采用零基整数作为数组索引之外，我们还可以采用任意字符串作为其索引，但是作为索引的字符串需要和数组元素值一样存在于 **ValueProvider** 的数据源中。索引数据项名称为“`index`”，并且与数组元素数据项具有相同的前缀。同样以上面这个参数类型为字符串数组的 **Action** 方法为例，可以通过提交具有如下内容的表单来调用这个 **Action** 方法并为之提供相应的参数值。

```
<input name="index" type="hidden" value="first" />
<input name="index" type="hidden" value="second" />
<input name="index" type="hidden" value="third" />

<input name="[first]" type="text" value="foo" />
<input name="[second]" type="text" value="bar" />
<input name="[third]" type="text" value="baz" />
```

被提交表单中三个文本框的值将会绑定到目标 **Action** 方法的字符串参数 `array`。它们通过基于字符串的索引进行命名（“`[first]`”、“`[second]`”和“`[third]`”），而作为索引的字符串通过 `hidden` 元素和作为参数绑定的数据一并提交，这些用于定义索引字符串的 `hidden` 元素统一命名为“`index`”。

现在我们对自定义的 **DefaultModelBinder** 进行进一步的完善，使基于数组的绑定支持两种索引。我们先定义了如下一个 `GetIndexes` 方法返回基于指定前缀的索引列表，输出参数 `numericIndex` 表示返回的索引是否是“零基索引”。针对文字的索引通过 **ValueProvider** 的 `GetValue` 方法获取，而指定的 `Key` 为“`{Prefix}.Index`”。如果这样的索引找不到，**DefaultModelBinder** 才会调用 `GetZeroBasedIndexes` 方法返回零基索引。

```
public class DefaultModelBinder : IModelBinder
{
    //其他成员
    private IEnumerable<string> GetIndexes(string prefix,
        IValueProvider valueProvider, out bool numericIndex)
    {
        string key = string.IsNullOrEmpty(prefix) ? "index" : prefix + "."
            + "index";
        ValueProviderResult result = valueProvider.GetValue(key);
        if (null != result)
        {
            string[] indexes = result.ConvertTo(typeof(string[])) as string[];
            if (null != indexes)
            {
                numericIndex = false;
            }
        }
    }
}
```

```

        return indexes;
    }
}
numericIndex = true;
return GetZeroBasedIndexes();
}

private static IEnumerable<string> GetZeroBasedIndexes()
{
    int iteratorVariable = 0;
    while (true)
    {
        yield return iteratorVariable.ToString();
        iteratorVariable++;
    }
}
}

```

接下来我们对用于进行数组绑定的 `GetArrayModel` 方法作进一步的完善。在该方法中，我们先调用另一个 `GetListModel` 方法将最终需要绑定到数组对象的所有元素保存到一个 `List<Object>` 对象中，然后根据目标数组元素类型创建一个具有与该列表具有相同长度的数组对象，并将列表的对象拷贝到该数组中。至于针对 `GetListModel` 方法的实现，我们先将前面介绍的针对同名数据项绑定的数组元素添加到这个列表中，然后遍历通过调用 `GetIndexes` 获取索引列表，将索引作为 `Key` 递归地调用 `GetModel` 方法得到目标数组对象的元素，并将其添加到列表中。

```

public class DefaultModelBinder : IModelBinder
{
    //其他成员
    protected virtual object GetArrayModel(ControllerContext controllerContext,
        Type modelType, IValueProvider valueProvider, string prefix)
    {
        List<object> list = GetListModel(controllerContext, modelType,
            modelType.GetElementType(), valueProvider, prefix);
        object[] array = (object[])Array.CreateInstance(
            modelType.GetElementType(), list.Count);
        list.CopyTo(array);
        return array;
    }

    private List<object> GetListModel(ControllerContext controllerContext,
        Type modelType, Type elementType, IValueProvider valueProvider,
        string prefix)
    {
        List<object> list = new List<object>();
        if (!string.IsNullOrEmpty(prefix) &&
            valueProvider.ContainsPrefix(prefix))
        {
            ValueProviderResult result = valueProvider.GetValue(prefix);
            if (null != result)
            {
                IEnumerable enumerable = result.ConvertTo(modelType)
                    as IEnumerable;
                foreach (var value in enumerable)
                {

```

```

        list.Add(value);
    }
}
}
bool numericIndex;
IEnumerable<string> indexes = GetIndexes(prefix, valueProvider,
    out numericIndex);
foreach (var index in indexes)
{
    string indexPrefix = prefix + "[" + index + "]";
    if (!valueProvider.ContainsPrefix(indexPrefix) && numericIndex)
    {
        break;
    }
    list.Add(GetModel(controllerContext, elementType, valueProvider,
        indexPrefix));
}
return list;
}
}

```

同样通过我们的实例来演示针对索引的数组绑定,为此我们对 HomeController 进行了如下修改: DemoAction 具有一个 Contact 数组类型的参数 contacts, 在该方法中将该数组对象的相关信息封装到一个字典对象中并作为 Model 呈现在默认的 View 中。通过 CreateValueProvider 方法返回的 NameValueCollectionValueProvider 以“零基索引”的形式提供了两个 Contact 对象的数据。提供的数据采用参数名称 contacts 作为前缀,按照前面介绍的“剔除前缀的后备 Model 绑定机制”,这个前缀在这里是可以省掉的。

```

public class HomeController : DemoController
{
    //其他成员
    protected override IValueProvider CreateValueProvider()
    {
        NameValueCollection dataSource = new NameValueCollection();

        dataSource.Add("contacts[0].Name", "张三");
        dataSource.Add("contacts[0].PhoneNo", "123456789");
        dataSource.Add("contacts[0].EmailAddress", "zhangsan@gmail.com");
        dataSource.Add("contacts[0].Address.Province", "江苏");
        dataSource.Add("contacts[0].Address.City", "苏州");
        dataSource.Add("contacts[0].Address.District", "工业园区");
        dataSource.Add("contacts[0].Address.Street", "星湖街 328 号");

        dataSource.Add("contacts[1].Name", "李四");
        dataSource.Add("contacts[1].PhoneNo", "987654321");
        dataSource.Add("contacts[1].EmailAddress", "lisi@gmail.com");
        dataSource.Add("contacts[1].Address.Province", "江苏");
        dataSource.Add("contacts[1].Address.City", "苏州");
        dataSource.Add("contacts[1].Address.District", "工业园区");
        dataSource.Add("contacts[1].Address.Street", "金鸡湖路 328 号");

        return new NameValueCollectionValueProvider(dataSource,
            CultureInfo.CurrentCulture);
    }
}

```

```

public ActionResult DemoAction(Contact[] contacts)
{
    Dictionary<string, object> parameters = new Dictionary<string, object>();
    for (int i = 0; i < contacts.Length; i++)
    {
        string name = contacts[i].Name;
        string phoneNo = contacts[i].PhoneNo;
        string emailAddress = contacts[i].EmailAddress;
        string address = string.Format("{0}省{1}市{2}{3}",
            contacts[i].Address.Province, contacts[i].Address.City,
            contacts[i].Address.District, contacts[i].Address.Street);

        parameters.Add(string.Format("[{0}].Name", i), name);
        parameters.Add(string.Format("[{0}].PhoneNo", i), phoneNo);
        parameters.Add(string.Format("[{0}].EmailAddress", i), emailAddress);
        parameters.Add(string.Format("[{0}].Address", i), address);
    }
    return View("DemoAction", parameters);
}

```

程序运行之后会在浏览器中得到如图 5-18 所示的输出结果，可以看出 DemoAction 的 Contact 数组参数按照我们希望的方式得到了绑定。(S514)

Name	Value
[0].Name	张三
[0].PhoneNo	123456789
[0].EmailAddress	zhangsan@gmail.com
[0].Address	江苏省苏州市工业园区星湖街328号
[1].Name	李四
[1].PhoneNo	987654321
[1].EmailAddress	lisi@gmail.com
[1].Address	江苏省苏州市工业园区金鸡湖路328号

图 5-18 数组类型的 Model 绑定 (2)

上面这个例子演示了针对基零整数作为索引的数组绑定，基于文字的索引同样是支持的。如果我们将 HomeController 的 GetValueProvider 方法改写成如下的形式，程序运行后依然会得到相同的输出结果。(S515)

```

public class HomeController : DemoController
{
    protected override IValueProvider CreateValueProvider()
    {
        NameValueCollection dataSource = new NameValueCollection();

        dataSource.Add("contacts.index", "first");
        dataSource.Add("contacts.index", "second");
    }
}

```



```

        dataSource.Add("contacts[first].Name", "张三");
        dataSource.Add("contacts[first].PhoneNo", "123456789");
        dataSource.Add("contacts[first].EmailAddress", "zhangsan@gmail.com");
        dataSource.Add("contacts[first].Address.Province", "江苏");
        dataSource.Add("contacts[first].Address.City", "苏州");
        dataSource.Add("contacts[first].Address.District", "工业园区");
        dataSource.Add("contacts[first].Address.Street", "星湖街 328 号");

        dataSource.Add("contacts[second].Name", "李四");
        dataSource.Add("contacts[second].PhoneNo", "987654321");
        dataSource.Add("contacts[second].EmailAddress", "lisi@gmail.com");
        dataSource.Add("contacts[second].Address.Province", "江苏");
        dataSource.Add("contacts[second].Address.City", "苏州");
        dataSource.Add("contacts[second].Address.District", "工业园区");
        dataSource.Add("contacts[second].Address.Street", "金鸡湖路 328 号");

        return new NameValueCollectionValueProvider(dataSource,
            CultureInfo.CurrentCulture);
    }
}

```

5.4.4 集合

这里的集合指的是除数组和字典之外的所有实现 `IEnumerable<T>` 接口的类型, 和基于数组的 Model 绑定类似, `ValueProvider` 可以将多个同名的数据项作为集合的元素, 基于索引(整数和字符串)绑定在这里同样适用。我们对自定义的 `DefaultModelBinder` 进一步完善使之支持集合类型的 Model 绑定, 为此我们定义了如下一个 `GetCollectionModel` 方法。

```

public class DefaultModelBinder : IModelBinder
{
    // 其他成员
    protected virtual object GetCollectionModel(
        ControllerContext controllerContext, Type modelType,
        IValueProvider valueProvider, string prefix)
    {
        Type elementType = modelType.GetGenericArguments()[0];
        List<object> list = GetListModel(controllerContext, modelType,
            elementType, valueProvider, prefix);
        object model = CreateModel(modelType);
        ReplaceHelper.ReplaceCollection(elementType, model, list);
        return model;
    }
}

internal static class ReplaceHelper
{
    private static MethodInfo replaceCollectionMethod = typeof(ReplaceHelper)
        .GetMethod("ReplaceCollectionImpl",
            BindingFlags.Static | BindingFlags.NonPublic);

    public static void ReplaceCollection(Type elementType, object model,
        object list)
    {

```

```

        replaceCollectionMethod.MakeGenericMethod(
            new Type[] { elementType }).Invoke(null,
            new object[] { model, list });
    }
    private static void ReplaceCollectionImpl<T>(
        ICollection<T> model, IEnumerable list)
    {
        model.Clear();
        if (list != null)
        {
            foreach (object obj2 in list)
            {
                T item = (obj2 is T) ? ((T)obj2) : default(T);
                model.Add(item);
            }
        }
    }
}

```

在 `GetCollectionModel` 方法中，我们按照数组绑定的方式调用 `GetListModel` 方法得到包含所有集合元素的 `List<Object>` 对象，然后根据数据类型调用 `CreateModel` 方法创建一个空的集合对象，根据前面给出的对该方法的定义我们知道最终创建出来的是一个 `List<>` 对象，最后我们调用定义在类型 `ReplaceHelper` 中的辅助方法 `ReplaceCollection` 将包含 `List<Object>` 对象中的元素拷贝到这个空 `List<>` 对象中并其返回。

现在需要将针对 `GetCollectionModel` 方法的调用放到 `GetModel` 方法中。如下面的代码片段所示，我们定义了一个 `ExtractGenericInterface` 方法来辅助判断数据类型是否是一个实现了 `IEnumerable<>` 接口的集合类型。需要注意的是，由于数组也实现了 `IEnumerable<T>` 接口，所以需要将下面这段代码放到针对数组绑定的 `GetArrayModel` 方法调用之后，否则 `GetArrayModel` 方法将永远得不到执行。

```

public class DefaultModelBinder : IModelBinder
{
    //其他成员
    public virtual object GetModel(ControllerContext controllerContext,
        Type modelType, IValueProvider valueProvider,
        string key)
    {
        //其他操作
        Type enumerableType = ExtractGenericInterface(modelType,
            typeof(IEnumerable<>));
        if (null != enumerableType)
        {
            return GetCollectionModel(controllerContext, modelType,
                valueProvider, key);
        }
        //其他操作
        return valueProvider.GetValue(key).ConvertTo(modelType);
    }

    private Type ExtractGenericInterface(Type queryType, Type interfaceType)
    {
        Func<Type, bool> predicate = t => t.IsGenericType &&
            (t.GetGenericTypeDefinition() == interfaceType);
    }
}

```

```

        if (!predicate(queryType))
        {
            return queryType.GetInterfaces().FirstOrDefault<Type>(predicate);
        }
        return queryType;
    }
}

```

在上面的实例中我们演示了针对数组的绑定（S515），如果需要演示针对集合的绑定，只需要按照如下的方式将定义在 HomeController 中的 DemoAction 方法的参数从 Contact 数组类型改为 IEnumerable<Contact>即可，运行新的程序之后依然可以得到如图 5-18 所示的输出结果。

```

public class HomeController : DemoController
{
    //其他成员
    public ActionResult DemoAction(IEnumerable<Contact> contacts)
    {
        Contact[] contactArray = contacts.ToArray();
        Dictionary<string, object> parameters = new Dictionary<string, object>();
        for (int i = 0; i < contactArray.Length; i++)
        {
            string name = contactArray[i].Name;
            string phoneNo = contactArray[i].PhoneNo;
            string emailAddress = contactArray[i].EmailAddress;
            string address = string.Format("{0}省{1}市{2}{3}",
                contactArray[i].Address.Province, contactArray[i].Address.City,
                contactArray[i].Address.District,
                contactArray[i].Address.Street);

            parameters.Add(string.Format("[{0}].Name", i), name);
            parameters.Add(string.Format("[{0}].PhoneNo", i), phoneNo);
            parameters.Add(string.Format("[{0}].EmailAddress", i), emailAddress);
            parameters.Add(string.Format("[{0}].Address", i), address);
        }
        return View("DemoAction", parameters);
    }
}

```

5.4.5 字典

这里的字典指的是实现了接口 IDictionary<TKey,TValue>的类型，在 Model 绑定过程中基于字典类型的数据映射很好理解。

- 字典是一个 KeyValuePair<TKey, TValue>对象的集合，所以在字典元素这一级可以采用基于索引的匹配机制。
- KeyValuePair<TKey, TValue>是一个复杂类型，可以按照属性名称（Key 和 Value）进行匹配。

比如说作为某个 ValueProvider 数据源的 NameValueCollection 具有如表 5-5 所示的结构，它可以映射为一个 IDictionary<string, Contact>对象（Contact 对象作为 Value，其 Name 属性作为 Key）。

表 5-5 字典对象在 NameValueCollection 中的表示

Key	Value
[0].Key	张三
[0].Value.Name	123456789
[0].Value.EmailAddress	zhangsan@gmail.com
[1].Key	李四
[1].Value.Name	987654321
[1].Value.EmailAddress	lisi@gmail.com

现在我们对用于模拟默认 Model 绑定的自定义 DefaultModelBinder 作最后的完善，使之支持针对字典类型的 Model 绑定。如下面的代码片段所示，针对字典类型的 Model 绑定被定义在 GetDictionaryModel 方法中。

```
public class DefaultModelBinder : IModelBinder
{
    //其他成员
    protected virtual object GetDictionaryModel(
        ControllerContext controllerContext, Type modelType,
        IValueProvider valueProvider, string prefix)
    {
        List<KeyValuePair<object, object>> list =
            new List<KeyValuePair<object, object>>();

        bool numericIndex;
        IEnumerable<string> indexes = GetIndexes(prefix, valueProvider,
            out numericIndex);
        Type[] genericArguments = modelType.GetGenericArguments();
        Type keyType = genericArguments[0];
        Type valueType = genericArguments[1];

        foreach (var index in indexes)
        {
            string indexPrefix = prefix + "[" + index + "]";
            if (!valueProvider.ContainsPrefix(indexPrefix) && numericIndex)
            {
                break;
            }
            string keyPrefix = indexPrefix + ".Key";
            string valulePrefix = indexPrefix + ".Value";
            object key = GetModel(controllerContext, keyType,
                valueProvider, keyPrefix);
            object value = GetModel(controllerContext, valueType,
                valueProvider, valulePrefix);
            list.Add(new KeyValuePair<object, object>(key, value));
        }
        object model = CreateModel(modelType);
        ReplaceHelper.ReplaceDictionary(keyType, valueType, model, list);
        return model;
    }
}
```

与在进行数组/集合绑定过程调用 `GetListModel` 方法获取所有数组/集合元素的 `List<Object>` 的逻辑类似，我们将最终绑定到目标字典对象的所有元素添加到一个预先创建的 `List<KeyValuePair<object, object>>` 对象中。对于每个 `KeyValuePair<object, object>` 对象的 `Key` 和 `Value`，我们递归地调用 `GetModel` 方法来实现，作为参数的 `Key` 由指定的前缀添加上 “.Key” 和 “.Value” 字符组合而成。接下来通过调用 `CreateModel` 方法针对数据类型创建一个空字典对象（`Dictionary<>`），然后通过定义在 `ReplaceHelper` 中具有如下定义的 `ReplaceDictionary` 方法将 `List<KeyValuePair<object, object>>` 对象中的元素拷贝到字典对象中并返回。

```
internal static class ReplaceHelper
{
    //其他成员
    private static MethodInfo replaceDictionaryMethod = typeof(ReplaceHelper)
        .GetMethod("ReplaceDictionaryImpl", BindingFlags.Static
            | BindingFlags.NonPublic);

    public static void ReplaceDictionary(Type keyType, Type valueType,
        object dictionary, object newContents)
    {
        replaceDictionaryMethod.MakeGenericMethod(
            new Type[] { keyType, valueType }).Invoke(null,
            new object[] { dictionary, newContents });
    }

    private static void ReplaceDictionaryImpl<TKey, TValue>(
        IDictionary<TKey, TValue> dictionary,
        IEnumerable<KeyValuePair<object, object>> newContents)
    {
        dictionary.Clear();
        foreach (KeyValuePair<object, object> pair in newContents)
        {
            TKey key = (TKey)pair.Key;
            TValue local2 = (TValue)((pair.Value is TValue)
                ? pair.Value : default(TValue));
            dictionary[key] = local2;
        }
    }
}
```

我们通过如下的方式将针对 `GetDictionaryModel` 方法的调用放到 `GetModel` 中，值得一提的是，由于字典类型也实现了 `IEnumerable<T>` 接口，我们必须将下面这段代码放到针对集合绑定的 `GetCollectionModel` 之前，否则这段代码将永远得不到执行。

```
public class DefaultModelBinder : IModelBinder
{
    //其他成员
    public virtual object GetModel(ControllerContext controllerContext,
        Type modelType, IValueProvider valueProvider, string key)
    {
        //其他操作
        Type dictionaryType = ExtractGenericInterface(modelType,
```

```

        typeof(IDictionary<, >));
    if (null != dictionaryType)
    {
        return GetDictionaryModel(controllerContext, modelType,
            valueProvider, key);
    }
    //其他操作
}
}

```

我们照例使用现成的实例来演示针对字典类型的绑定，为此我们对 HomeController 进行了如下的改写：Action 方法 DemoAction 具有一个 IDictionary<string, Contact> 类型的参数，我们将该参数封装成另一个字典对象并作为 Model 对象呈现在默认的 View 中。在 CreateValueProvider 方法中创建 NameValueCollectionValueProvider 对象按照基于字典绑定规则添加了两个 Contact 对象的数据。

```

public class HomeController : DemoController
{
    //其他成员
    protected override IValueProvider CreateValueProvider()
    {
        NameValueCollection dataSource = new NameValueCollection();

        dataSource.Add("contacts.index", "first");
        dataSource.Add("contacts.index", "second");

        dataSource.Add("contacts[first].Key", "张三");
        dataSource.Add("contacts[first].Value.Name", "张三");
        dataSource.Add("contacts[first].Value.PhoneNo", "123456789");
        dataSource.Add("contacts[first].Value.EmailAddress",
            "zhangsan@gmail.com");
        dataSource.Add("contacts[first].Value.Address.Province", "江苏");
        dataSource.Add("contacts[first].Value.Address.City", "苏州");
        dataSource.Add("contacts[first].Value.Address.District", "工业园区");
        dataSource.Add("contacts[first].Value.Address.Street", "星湖街 328 号");

        dataSource.Add("contacts[second].Key", "李四");
        dataSource.Add("contacts[second].Value.Name", "李四");
        dataSource.Add("contacts[second].Value.PhoneNo", "987654321");
        dataSource.Add("contacts[second].Value.EmailAddress", "lisi@gmail.com");
        dataSource.Add("contacts[second].Value.Address.Province", "江苏");
        dataSource.Add("contacts[second].Value.Address.City", "苏州");
        dataSource.Add("contacts[second].Value.Address.District", "工业园区");
        dataSource.Add("contacts[second].Value.Address.Street", "金鸡湖路 328 号");

        return new NameValueCollectionValueProvider(dataSource,
            CultureInfo.CurrentCulture);
    }

    public ActionResult DemoAction(IDictionary<string, Contact> contacts)
    {
        var contactArray = contacts.ToArray();
        Dictionary<string, object> parameters = new Dictionary<string, object>();
        foreach (var item in contacts)

```

```

    {
        string address = string.Format("{0}省{1}市{2}{3}",
            item.Value.Address.Province, item.Value.Address.City,
            item.Value.Address.District, item.Value.Address.Street);
        parameters.Add(string.Format("contacts[{0}].Name", item.Key),
            item.Value.Name);
        parameters.Add(string.Format("contacts[{0}].PhoneNo",
            item.Key), item.Value.PhoneNo);
        parameters.Add(string.Format("contacts[{0}].EmailAddress",
            item.Key), item.Value.EmailAddress);
        parameters.Add(string.Format("contacts[{0}].Address", item.Key),
            address);
    }

    return View("DemoAction", parameters);
}
}

```

该程序运行之后会在浏览器中呈现出如图 5-19 所示的输出结果，可以看到 Action 方法 DemoAction 的参数按照我们期望的方式完成了绑定。（S517）



Name	Value
contacts["张三"].Name	张三
contacts["张三"].PhoneNo	123456789
contacts["张三"].EmailAddress	zhangsan@gmail.com
contacts["张三"].Address	江苏省苏州市工业园区星湖街328号
contacts["李四"].Name	李四
contacts["李四"].PhoneNo	987654321
contacts["李四"].EmailAddress	lisi@gmail.com
contacts["李四"].Address	江苏省苏州市工业园区金鸡湖路328号

图 5-19 字典类型的 Model 绑定

前面我们通过自定义的 DefaultModelBinder 模拟了 ASP.NET MVC 默认的 Model 绑定实现机制。由于篇幅所限，自定义的这个 ModelBinder 不可能涵盖所有的细节（比如通过 BindAttribute 特性的 Exclude 和 Include 属性如何控制数据成员是否参与绑定就没有涉及），但是它基本上能够反映定义在 ASP.NET MVC 中 DefaultModelBinder 的实现逻辑。

本章小结

针对某个请求的处理主要体现在对目标 Action 方法的执行，而 Action 方法执行的前提在于能够预先提供相应的参数列表，所以旨在为目标 Action 方法提供参数列表的 Model 绑定在整个 ASP.NET MVC 中的地位便可想而知。

在整个 Model 绑定过程中，ValueProvider 完成了针对数据的提供。一个 ValueProvider 具有

一个内部数据容器，可以是一个 `NameValueCollection` 对象，也可能是一个 `Dictionary<TKey, TValue>` 对象，它们对应着两种基本的 `ValueProvider`，即 `NameValueCollectionValueProvider` 和 `DictionaryValueProvider`。`FormValueProvider` 和 `QueryStringValueProvider` 是两种典型并且常用的 `NameValueCollectionValueProvider`，而 `DictionaryValueProvider` 则包括 `RouteDataValueProvider`、`HttpFileCollectionValueProvider` 和 `ChildActionValueProvider`。

`ValueProvider` 通过对应的 `ValueProviderFactory` 来创建，如果 ASP.NET MVC 默认的提供机制不能满足需要，可以自定义 `ValueProviderFactory` 来创建针对某种数据来源（比如 HTTP 报头）的 `ValueProvider`。自定义的 `ValueProviderFactory` 通过静态类型 `ValueProviderFactories` 进行注册。

`ModelBinder` 是整个 Model 绑定体系的核心，它通过 `ValueProvider` 提供的数组构建一个作为 Action 方法参数的对象。ASP.NET MVC 会采用 `DefaultModelBinder` 来完成 Model 绑定工作，在本章的最后一节我们用了大量的篇幅通过实例演示的方式介绍了 `DefaultModelBinder` 针对简单类型、复杂类型、数组、集合和字典类型的 Model 绑定。

如果默认的 `DefaultModelBinder` 不能满足要求，可以自定义 `ModelBinder`。自定义的绑定可以通过 `ModelBinderAttribute` 或者自定义的 `CustomModelBinderAttribute` 特性应用到 Action 方法参数或者数据类型上，以实现针对指定参数或者目标数据类型的 Model 绑定的控制。我们也可以通过自定义 `ModelBinderProvider` 来作为针对某种数据类型的 `ModelBinder` 提供者，自定义 `ModelBinderProvider` 通过静态类型 `ModelBinderProviders` 进行注册。除此之外还可以通过静态类型 `ModelBinders` 将自定义的 `ModelBinder` 注册到目标数据类型上。

第 6 章 Model 的验证

Action 方法在执行之前需要通过 Model 验证机制确保提供参数的有效性，Model 验证是伴随着 Model 绑定进行的。ASP.NET MVC 默认采用声明式的验证规则定义方式，应用在数据类型及其属性上的验证特性最终成为 Model 元数据的一部分。除了必需的服务端验证之外，ASP.NET MVC 利用 JavaScript 实现客户端验证。

6.1 ModelValidator 与 ModelValidatorProvider

很多 ASP.NET MVC 的书籍在介绍 Model 验证这部分的时候都会先为我们列出一系列的验证特性，但是本章旨在剖析 ASP.NET MVC 的 Model 验证系统的总体设计和运行原理，所以验证特性不是本质的东西，而用于真正实施验证的 ModelValidator 才是最核心的组件，所以我们先来了解一下 ModelValidator 及其提供机制。

6.1.1 ModelValidator

ASP.NET MVC 中的 ModelValidator 继承自抽象类型 System.Web.Mvc.ModelValidator。如下面的代码片段所示，该类型具有一个布尔类型的只读属性 IsRequired，表示该 ModelValidator 是否对目标数据进行“必需性”验证（即被验证的数据成员具有一个具体的值），该属性默认返回 False。GetClientValidationRules 返回一个元素类型为 System.Web.Mvc.ModelClientValidationRule 的集合，而 ModelClientValidationRule 是对客户端验证规则的封装，我们会在客户端验证部分对其进行详细介绍。

```
public abstract class ModelValidator
{
    //其他成员
    public virtual IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
    public abstract IEnumerable<ModelValidationResult> Validate(
        object container);

    public virtual bool IsRequired { get; }
}
```

针对目标数据的验证是通过调用 Validate 方法来完成的，该方法的输入参数 container 表示的正是被验证的对象。由于 ASP.NET MVC 下的 Model 验证仅仅针对自定义类型，所以被验证的对象是一个“容器”对象，这一点可以通过参数名称“container”体现出来。Validate 方法表示验证结果的返回值并不是一个简单的布尔值，而是一个元素类型为具有如下定义的 System.Web.Mvc.ModelValidationResult 对象集合。

```
public class ModelValidationResult
{
    public string MemberName { get; set; }
    public string Message { get; set; }
}
```

ModelValidationResult 具有两个字符串类型属性 MemberName 和 Message，前者代表被验证数据成员的名称，后者表示错误消息。一般来说，如果 ModelValidationResult 对象来源于针对容器对象本身的验证，它的 MemberName 属性为空字符串；而对于容器对象某个属性的验证来说，属性名称作为返回的 ModelValidationResult 对象的 MemberName 属性。

`ModelValidationResult` 集合只有在验证失败的情况下才会返回。如果被验证数据对象符合所有的验证规则，`Validate` 方法会直接返回 `Null` 或者一个空 `ModelValidationResult` 集合。值得一提的是，我们有时候会用 `System.ComponentModel.DataAnnotations.ValidationResult` 的静态只读字段 `Success` 表示成功通过验证的结果，实际上该字段的值就是 `Null`。

```
public class ValidationResult
{
    //其他成员
    public static readonly ValidationResult Success;
}
```

DataAnnotationsModelValidator

对 ASP.NET MVC 稍微了解的读者应该知道，我们可以在数据类型及其属性上应用相应的验证特性（比如 `RequiredAttribute`、`RangeAttribute` 和 `RegularExpressionAttribute` 等）来定义验证规则。但这仅仅是 `Model` 验证的其中一种解决方案而已，基于数据注解验证特性这种声明式验证解决方案最终通过 `System.Web.Mvc.DataAnnotationsModelValidator` 来实现。由于它代表一种最为常用的 `Model` 验证方案，在本章后续部分我们将对它进行深入的介绍。

ClientModelValidator

`ClientModelValidator` 是定义在程序集 `System.Web.Mvc.dll` 中的内部类型，从其命名就可以看出它仅仅用于客户端验证。如下面的代码片段所示，我们通过调用构造函数创建一个 `ClientModelValidator` 的时候，不仅需要指定描述被验证对象类型的 `ModelMetadata` 和当前 `ControllerContext`，还需要以字符串的形式指定验证类型和错误消息。

```
internal class ClientModelValidator : ModelValidator
{
    public ClientModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext, string validationType,
        string errorMessage);

    public sealed override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
    public sealed override IEnumerable<ModelValidationResult> Validate(
        object container);
}
```

由于 `ClientModelValidator` 仅限于客户端验证，用于实现服务端验证的 `Validate` 方法总是返回一个空的 `ModelValidationResult` 集合（表示验证成功），它的 `GetClientValidationRules` 方法返回的是一个元素类型为 `System.Web.Mvc.ModelClientValidationRule` 的集合。`ModelClientValidationRule` 表示客户端验证规则，它们最终将会出现在 HTML 中辅助 JavaScript 验证框架（比如 jQuery）实施客户端验证。

`ClientModelValidator` 具有两个继承者，分别是针对数值类型和日期类型验证的 `NumericModelValidator` 和 `DateModelValidator`。如下面的代码片段所示，这两个 `ClientModelValidator` 用于表示验证类型的字符串分别是“number”和“date”，而表示错误消息的字符串是从内部维护的资源文件中获取的。这实际上带来了一个问题，就是我们无法对错误消息进行定制。

```
internal sealed class NumericModelValidator : ClientModelValidator
{
    public NumericModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext)
        : base(metadata, controllerContext, "number",
            ClientDataTypeModelValidatorProvider.GetFieldMustBeNumericResource(
                controllerContext))
    {
    }
}

internal sealed class DateModelValidator : ClientModelValidator
{
    public DateModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext)
        : base(metadata, controllerContext, "date",
            ClientDataTypeModelValidatorProvider.GetFieldMustBeDateResource(
                controllerContext))
    {
    }
}
```

DataErrorInfoModelValidator

在 `System.ComponentModel` 命名空间下定义了一个名为 `IDataErrorInfo` 的接口，它提供了一种标准的错误信息定制方式。如下面的代码片段所示，`IDataErrorInfo` 具有两个成员，只读属性 `Error` 用于获取基于自身的错误消息，而只读索引用于返回指定数据成员的错误消息。

```
public interface IDataErrorInfo
{
    string Error { get; }
    string this[string columnName] { get; }
}
```

如果被验证对象的类型实现了 `IDataErrorInfo` 接口，意味着我们可以通过上述两个成员获取到相应的错误消息。`ASP.NET MVC` 为此专门定义了两个对应的 `ModelValidator`，即 `DataErrorInfoClassModelValidator` 和 `DataErrorInfoPropertyModelValidator`。不过它们都是内部类型，我们不能直接使用它们进行 `Model` 验证。

从名称也可以看出来，`DataErrorInfoClassModelValidator` 实现针对容器对象本身的验证，它通过 `Error` 属性获取相应的验证错误消息。`DataErrorInfoPropertyModelValidator` 则致力于

容器对象数据成员（属性）的验证，它根据当前属性名称从索引中获取对应的验证错误消息。

ValidatableObjectAdapter

在 `System.ComponentModel.DataAnnotations` 命名空间下定义了一个 `IValidatableObject` 接口，它代表另外一种验证的模式，笔者将其称为“自我验证”，即数据对象自行实现针对自身的验证。如下面的代码片段所示，针对自身的验证实现在 `Validate` 方法中。该方法的参数不是被验证的对象，而是通过 `ValidationContext` 类型表示的验证上下文，返回值则是一个 `ValidationResult` 对象的集合。`ValidationContext` 和 `ValidationResult` 这两个类型均定义在 `System.ComponentModel.DataAnnotations` 命名空间下。

```
public interface IValidatableObject
{
    IEnumerable<ValidationResult> Validate(ValidationContext validationContext);
}
```

ASP.NET MVC 定义了专门的 `ModelValidator` 对实现了 `IValidatableObject` 接口的数据类型实施验证，对应的类型为 `System.Web.Mvc.ValidatableObjectAdapter`。由于被验证对象本身已经将验证逻辑实现在了 `Validate` 方法中，所以 `ValidatableObjectAdapter` 只需要调用该方法并将返回的验证结果从 `ValidationResult` 类型转换成 `ModelValidationResult` 类型即可。

```
public class ValidatableObjectAdapter : ModelValidator
{
    public ValidatableObjectAdapter(ModelMetadata metadata,
        ControllerContext context);
    public override IEnumerable<ModelValidationResult> Validate(
        object container);
}
```

6.1.2 ModelValidatorProvider

通过前面章节的介绍我们知道，ASP.NET MVC 大都采用 `Provider` 的模式来提供相应的组件，比如描述 `Model` 元数据的 `ModelMetadata` 通过对应的 `ModelMetadataProvider` 来提供，实现 `Model` 绑定的 `ModelBinder` 则可以通过对应的 `ModelBinderProvider` 来提供，用于实现 `Model` 验证的 `ModelValidator` 也不例外，它对应的提供者是 `ModelValidatorProvider`。`ModelValidatorProvider` 继承自具有如下定义的 `System.Web.ModelBinding.ModelValidatorProvider` 抽象类。

```
public abstract class ModelValidatorProvider
{
    protected ModelValidatorProvider();
    public abstract IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context);
}
```

如上面的代码片段所示，GetValidators 方法具有两个参数，一个是用于描述被验证类型或者属性元数据的 ModelMetadata 对象，另一个是当前 ControllerContext，该方法返回的是一个元素类型为 ModelValidator 的集合。

DataAnnotationsModelValidatorProvider

DataAnnotationsModelValidator 提供了我们最常用的基于验证特性的声明式 Model 验证，它对应的 ModelValidatorProvider 类型为 System.Web.Mvc.DataAnnotationsModelValidatorProvider。如下面的代码片段所示，DataAnnotationsModelValidatorProvider 并没有直接继承 ModelValidatorProvider，它的父类是另一个名为 System.Web.Mvc.AssociatedValidatorProvider 的抽象类。

```
public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    //其他成员
    public DataAnnotationsModelValidatorProvider();
    protected override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context,
        IEnumerable<Attribute> attributes);
}
```

AssociatedValidatorProvider 类型名称中所谓的“关联（Association）”代表的就是关联的特性，它利用从 ModelMetadata 提取的特性来构建相应的 ModelValidator 对象。如下面的代码片段所示，AssociatedValidatorProvider 定义了一个受保护的虚方法 GetTypeDescriptor，它用于获取指定类型的描述对象（其类型实现了 ICustomTypeDescriptor 接口）。

```
public abstract class AssociatedValidatorProvider : ModelValidatorProvider
{
    public sealed override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context);

    protected virtual ICustomTypeDescriptor GetTypeDescriptor(Type type);
    protected abstract IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context,
        IEnumerable<Attribute> attributes);
}
```

如果被验证对象是一个容器对象的某个属性（这可以通过 ModelMetadata 分析出来），AssociatedValidatorProvider 会调用 GetTypeDescriptor 方法得到容器类型描述对象，并进一步通过 ModelMetadata 提供的属性名称得到应用在属性及其属性类型上的特性列表。对于针对“根容器对象”的验证来说，它会调用 GetTypeDescriptor 得到被验证类型的描述对象，并据此得到应用在数据类型上的特性列表。得到的特性列表最终作为参数调用受保护的抽象方法 GetValidators，该方法根据提供的特性创建相应的 ModelValidator 列表，这个列表就是公有 GetValidators 方法的返回值。

DataAnnotationsModelValidatorProvider 正是通过实现这个受保护的抽象方法

GetValidators 完成了针对 DataAnnotationsModelValidator 的提供。具体来说，它们从提供的特性列表中筛选出继承自 ValidationAttribute 的验证特性，并根据它们创建相应的 DataAnnotationsModelValidator 对象。

ClientDataTypeModelValidatorProvider

针对数字/日期类型客户端验证的 NumericModelValidator 和 DateModelValidator 最终是通过具有如下定义的 System.Web.Mvc.ClientDataTypeModelValidatorProvider 来提供的。在实现的 GetValidators 方法中，它会根据指定 ModelMetadata 判断被验证类型是否属于数字/DateTime 类型，如果是则直接返回一个包含单个 NumericModelValidator 或者 DateModelValidator 对象的 ModelValidator 集合。在这里被视为数字的类型包括 byte、sbyte、short、ushort、int、uint、long、ulong、float、double 和 decimal 等。

```
public class ClientDataTypeModelValidatorProvider : ModelValidatorProvider
{
    public ClientDataTypeModelValidatorProvider();
    public override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context);
}
```

DataErrorInfoModelValidatorProvider

旨在对实现了 IDataErrorInfo 接口的数据实施验证的两个 DataErrorInfoModelValidator（即 DataErrorInfoClassModelValidator 和 DataErrorInfoPropertyModelValidator），最终是通过具有如下定义的 System.Web.Mvc.DataErrorInfoModelValidatorProvider 来提供的。在实现的 GetValidators 方法中，如果被验证数据类型实现了 IDataErrorInfo 接口，它会基于指定的 ModelMetadata 和 ControllerContext 创建一个 DataErrorInfoClassModelValidator 对象置于返回的 ModelValidator 集合中。

如果被验证的对象是容器对象的某个属性，并且容器对象的类型（不是属性类型）实现了 IDataErrorInfo 接口，该方法返回的 ModelValidator 集合中还会包含一个基于指定 ModelMetadata 和 ControllerContext 创建的数据 ErrorInfoPropertyModelValidator 对象。

```
public class DataErrorInfoModelValidatorProvider : ModelValidatorProvider
{
    public DataErrorInfoModelValidatorProvider();
    public override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context);
}
```

为了让读者更深刻地了解实现在 DataErrorInfoModelValidator 中的验证机制，以及实现在 DataErrorInfoModelValidatorProvider 中对它的提供机制，我们来演示一个简单的实例。在一个 ASP.NET MVC 应用中定义了如下一个实现了 IDataErrorInfo 接口的 Contact 类型，假设

提供的 `Contact` 对象总是不能通过验证，并为整个 `Contact` 容器对象和它的 4 个属性成员定义了相应的验证错误消息。

```
public class Contact: IDataErrorInfo
{
    public string Error
    {
        get { return "无效联系人! ";}
    }

    public string this[string columnName]
    {
        get
        {
            switch (columnName)
            {
                case "Name"           : return "姓名是必需的! ";
                case "PhoneNo"        : return "电话号码格式错误! ";
                case "EmailAdderss"   : return "无效的电子邮箱地址! ";
                default                : return null;
            }
        }
    }

    public string Name { get; set; }
    public string PhoneNo { get; set; }
    public string EmailAdderss { get; set; }
}
```

然后创建了如下一个 `HomeController`，辅助方法 `GetValidators` 根据指定的数据类型和 `ModelValidatorProvider` 对象创建了一个 `ModelValidator` 集合，该集合同时包括针对数据类型本身和它所有属性成员的 `ModelValidator`。在默认的 `Action` 方法 `Index` 中我们将创建的 `DataErrorInfoModelValidatorProvider` 对象作为参数调用 `GetValidators` 方法，并将得到的 `ModelValidator` 集合 `Model` 呈现在默认的 `View` 中。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ModelValidatorProvider validatorProvider =
            new DataErrorInfoModelValidatorProvider();
        return View(GetValidators(typeof(Contact), validatorProvider));
    }

    private IEnumerable<ModelValidator> GetValidators(Type dataType,
        ModelValidatorProvider validatorProvider)
    {
        ModelMetadata metadata = ModelMetadataProviders.Current
            .GetMetadataForType(null, dataType);

        foreach (var validator in validatorProvider.GetValidators(metadata,
            ControllerContext))
        {

```

```

        yield return validator;
    }

    foreach (var propertyMetadata in metadata.Properties)
    {
        foreach (var validator in validatorProvider
            .GetValidators(propertyMetadata, ControllerContext))
        {
            yield return validator;
        }
    }
}
}

```

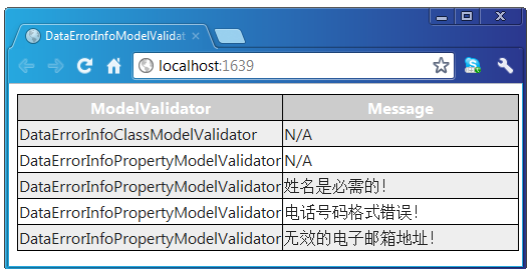
如下所示的是 Action 方法 Index 对应 View 的定义，可以看出这是一个 Model 类型为 IEnumerable<ModelValidator> 的强类型 View。在该 View 中我们使用每一个 ModelValidator 对创建的 Contact 实施验证，并将 ModelValidator 的类型和验证错误消息通过表格的形式呈现出来。

```

@model IEnumerable<ModelValidator>
<html>
    <head>
        <title>DataErrorInfoModelValidatorProvider</title>
    </head>
    @{
        Contact contact = new Contact();
    }
    <body>
        <table>
            <tr><th>ModelValidator</th><th>Message</th></tr>
            @foreach (var validator in Model)
            {
                ModelValidationResult[] results =
                    validator.Validate(contact).ToArray();
                string firstMessage = (results.Any() ?
                    results.First().Message : "N/A");
                <tr>
                    <td rowspan="@results.Length">@validator.GetType().Name</td>
                    <td>@firstMessage</td></tr>
                for (int i = 1; i < results.Length; i++)
                {
                    <tr><td>@results[i].Message</td></tr>
                }
            }
        </table>
    </body>
</html>

```

上面的程序运行之后会在浏览器中呈现如图 6-1 所示的输出结果。可以看到通过 DataErrorInfoModelValidatorProvider 创建了 1 个 DataErrorInfoClassModelValidator 和 4 个 DataErrorInfoPropertyModelValidator，前者针对 Contact 类型本身，后者针对定义在 Contact 中的 4 个属性。(S601)



ModelValidator	Message
DataErrorInfoClassModelValidator	N/A
DataErrorInfoPropertyModelValidator	N/A
DataErrorInfoPropertyModelValidator	姓名是必需的!
DataErrorInfoPropertyModelValidator	电话号码格式错误!
DataErrorInfoPropertyModelValidator	无效的电子邮箱地址!

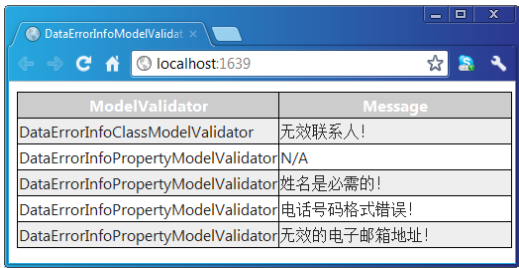
图 6-1 针对 DataErrorInfoModelValidator 的 Model 验证 (1)

针对属性的 DataErrorInfoPropertyModelValidator 能够按照我们希望的方式对指定的 Contact 对象实施验证,但是针对容器类型对象本身的 DataErrorInfoClassModelValidator 则不能,因为当 DataErrorInfoClassModelValidator 的 Validate 方法被调用的时候,它并不会真正去验证通过参数指定的容器对象,而是去验证包含在初始化该 DataErrorInfoClassModelValidator 对象时指定的 ModelMetadata 的 Model 属性。由于我们提供的 ModelMetadata 对象的 Model 属性值为 Null,所以 DataErrorInfoClassModelValidator 会直接返回一个空的 ModelValidationResult 集合。

为了验证这一点,我们可以对定义在 HomeController 中的 GetValidators 方法作如下的改动:在创建针对 ModelMetadata 时指定了一个创建 Contact 的表达式,该表达式执行后得到的对象将作为 ModelMetadata 的 Model 属性。

```
public class HomeController : Controller
{
    //其他成员
    private IEnumerable<ModelValidator> GetValidators(Type dataType,
        ModelValidatorProvider validatorProvider)
    {
        ModelMetadata metadata = ModelMetadataProviders.Current
            .GetMetadataForType(()=>new Contact(), dataType);
        //其他操作
    }
}
```

再次运行程序后会在浏览器中得到如图 6-2 所示的输出结果,可以清楚地看到针对 DataErrorInfoClassModelValidator 的验证错误消息(“无效联系人!”)出现了。(S602)



ModelValidator	Message
DataErrorInfoClassModelValidator	无效联系人!
DataErrorInfoPropertyModelValidator	N/A
DataErrorInfoPropertyModelValidator	姓名是必需的!
DataErrorInfoPropertyModelValidator	电话号码格式错误!
DataErrorInfoPropertyModelValidator	无效的电子邮箱地址!

图 6-2 针对 DataErrorInfoModelValidator 的 Model 验证 (2)

6.1.3 ModelValidatorProviders

通过静态类型 `System.Web.Mvc.ModelValidatorProviders` 对 `ModelValidatorProvider` 进行注册。如下面的代码片段所示, `ModelValidatorProviders` 具有一个静态只读属性 `Providers`, 对应的类型为 `ModelValidatorProviderCollection`, 表示基于整个 Web 应用范围的全局 `ModelValidatorProvider` 集合。

```
public static class ModelValidatorProviders
{
    public static ModelValidatorProviderCollection Providers { get; }
}

public class ModelValidatorProviderCollection :
    Collection<ModelValidatorProvider>
{
    public ModelValidatorProviderCollection();
    public ModelValidatorProviderCollection(IList<ModelValidatorProvider> list);
    public IEnumerable<ModelValidator> GetValidators(ModelMetadata metadata,
        ControllerContext context);
}

public class ModelMetadata
{
    //其他成员
    public virtual IEnumerable<ModelValidator> GetValidators(
        ControllerContext context);
}
```

`ModelValidatorProviderCollection` 定义了一个 `GetValidators` 方法返回一个 `ModelValidator` 列表, 集合中每个 `ModelValidatorProvider` 创建的 `ModelValidator` 会包含在该列表中。`ModelMetadata` 的 `GetValidators` 方法返回的 `ModelValidator` 列表正是通过 `ModelValidatorProviders` 的静态属性 `Providers` 创建的。

当 `ModelValidatorProviders` 类型被加载的时候, 它会创建三个 `ModelValidatorProvider` 并添加到通过静态属性 `Providers` 表示的 `ModelValidatorProvider` 集合之中, 而这三个 `ModelValidatorProvider` 对应的类型分别为 `DataAnnotationsModelValidatorProvider`、`ClientDataTypeModelValidatorProvider` 和 `DataErrorInfoPropertyModelValidator`。

如果我们需要注册一个自定义 `ModelValidatorProvider`, 可以直接将相应的对象添加到 `ModelValidatorProviders` 的 `Providers` 列表中。如果需要采用自定义 `ModelValidatorProvider` 来替换掉现有的 `ModelValidatorProvider`, 比如我们创建了一个扩展的 `DataAnnotationsModelValidatorProvider`, 还需要将现有的 `ModelValidatorProvider` 从该列表中移除。

上面我们介绍了用于进行 Model 验证的 `ModelValidator`, 用于提供 `ModelValidator` 的 `ModelValidatorProvider`, 以及用于注册 `ModelValidatorProvider` 的 `ModelValidatorProviders`, 整个 `ModelValidator` 的提供系统以此三类组件为核心, 图 6-3 所示的 UML 体现了它们之间的关系。

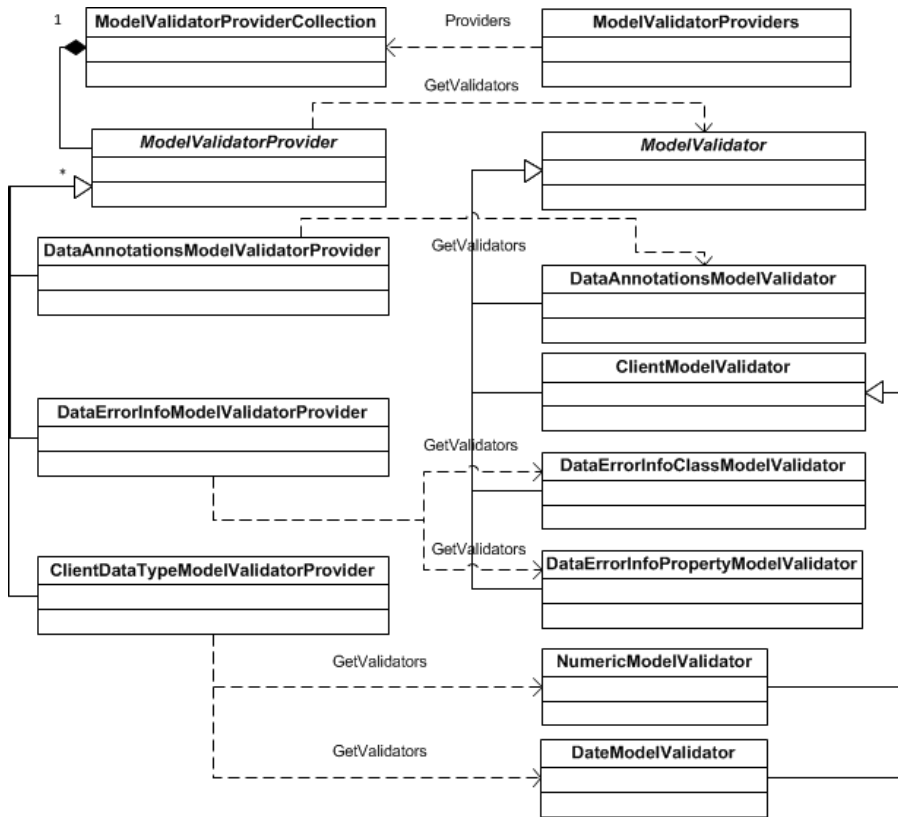


图 6-3 ModelValidator、ModelValidatorProvider 和 ModelValidatorProviders 之间的关系

CompositeModelValidator

虽然 CompositeModelValidator 仅仅是定义在程序集 System.Web.Mvc.dll 中的一个私有类型，但是它在 ASP.NET MVC 的 Model 验证系统中具有重要的地位，可以说真正用于 Model 验证的 ModelValidator 就是这么一个对象。

```
private class CompositeModelValidator : ModelValidator
{
    public CompositeModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext);
    public override IEnumerable<ModelValidationResult> Validate(
        object container);
}
```

CompositeModelValidator 实际上并不是一个真正对 Model 对象实施验证的 ModelValidator，它是一系列 ModelValidator 的组合，它根据基于数据本身类型及其属性的 ModelMetadata 动态地获取相应的 ModelValidator（通过调用 ModelMetadata 的 GetValidators 方法）对目标数据实施验证。抽象类 ModelValidator 具有一个静态的 GetModelValidator 方法，它根据指定的 ModelMetadata 和 ControllerContext 得到相应的 ModelValidator 对象。如下面的代码片段所示，该方法返回的正是一个 CompositeModelValidator 对象。

```

public abstract class ModelValidator
{
    //其他成员
    public static ModelValidator GetModelValidator(ModelMetadata metadata,
        ControllerContext context)
    {
        return new CompositeModelValidator(metadata, context);
    }
}

```

当 `CompositeModelValidator` 被用于验证一个容器对象的时候，会先验证其属性成员。针对容器对象自身的验证只有在所有属性值都通过验证的情况下才会进行。具体的逻辑是这样的：它通过调用描述容器类型的 `ModelMetadata` 对象的 `Properties` 属性得到所有针对属性的 `ModelMetadata` 对象，然后调用它们的 `GetValidators` 方法得到一组 `ModelValidator` 对相应的属性值实施验证，验证得到的 `ModelValidationResult` 被添加到最终返回的 `ModelValidationResult` 列表中。

如果在对所有属性实施验证之后该 `ModelValidationResult` 列表依然为空（所有的属性均成功通过验证），`CompositeModelValidator` 才会获取针对容器类型的 `ModelMetadata` 对象，并采用调用其 `GetValidators` 方法获取的 `ModelValidator` 列表对容器对象本身实施验证。表示验证结果的 `ModelValidationResult` 对象被添加到最终返回的列表中。

实例演示：CompositeModelValidator 采用的验证行为（S603, S604）

为了使读者对 `CompositeModelValidator` 的验证逻辑具有一个深刻的理解，我们来演示一个具体的实例。在一个 ASP.NET MV 应用中定义了如下一个名称为 `AlwaysFailsAttribute` 的验证特性。如下面的代码片段所示，重写的 `IsValid` 方法总是返回 `False`，意味着针对数据的验证总是会失败。我们还重写了只读属性 `TypeId`，让它真正能够唯一标识一个 `AlwaysFailsAttribute` 特性实例（具体原因我们会在本章后续部分予以介绍）。

```

[AttributeUsage( AttributeTargets.Class| AttributeTargets.Property)]
public class AlwaysFailsAttribute : ValidationAttribute
{
    private object typeId;
    public override bool IsValid(object value)
    {
        return false;
    }
    public override object TypeId
    {
        get { return typeId ?? (typeId = new object()); }
    }
}

```

我们将 `AlwaysFailsAttribute` 应用到表示联系人的 `Contact` 类型上。如下面的代码片段所示，在 `Contact` 和 `Address` 的类型和属性都应用了该特性，并且指定了相应的错误消息。

```

[AlwaysFails(ErrorMessage = "Contact")]
public class Contact
{

```

```

[AlwaysFails(ErrorMessage = "Contact.Name")]
public string Name { get; set; }

[AlwaysFails(ErrorMessage = "Contact.PhoneNo")]
public string PhoneNo { get; set; }

[AlwaysFails(ErrorMessage = "Contact.EmailAddress")]
public string EmailAddress { get; set; }

[AlwaysFails(ErrorMessage = "Contact.Address")]
public Address Address { get; set; }
}

[AlwaysFails(ErrorMessage = "Address")]
public class Address
{
    [AlwaysFails(ErrorMessage = "Address.Province")]
    public string Province { get; set; }

    [AlwaysFails(ErrorMessage = "Address.City")]
    public string City { get; set; }

    [AlwaysFails(ErrorMessage = "Address.District")]
    public string District { get; set; }

    [AlwaysFails(ErrorMessage = "Address.Street")]
    public string Street { get; set; }
}

```

我们创建了一个具有如下定义的 HomeController 类, 在 Action 方法 Index 中, 使用当前注册的 ModelMetadataProvider 创建了描述 Contact 类型的 ModelMetadata 对象, 然后将它和当前 ControllerContext 作为参数调用抽象类型 ModelValidator 的静态方法 GetValidator 创建一个 CompositeModelValidator 对象。我们利用该 CompositeModelValidator 来验证创建的 Contact 对象, 并将表示验证结果的 ModelValidationResult 列表作为 Model 呈现在默认的 View 中。

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        Address address = new Address
        {
            Province    = "江苏",
            City         = "苏州",
            District     = "工业园区",
            Street       = "星湖街 328 号"
        };

        Contact contact = new Contact
        {
            Name         = "张三",
            PhoneNo      = "123456789",
            EmailAddress = "zhangsan@gmail.com",
            Address       = address
        };

        ModelMetadata metadata = ModelMetadataProviders.Current

```

```

        .GetMetadataForType(() => contact, typeof(Contact));
        ModelValidator validator = ModelValidator.GetModelValidator(metadata,
            ControllerContext);
        return View(validator.Validate(contact));
    }
}

```

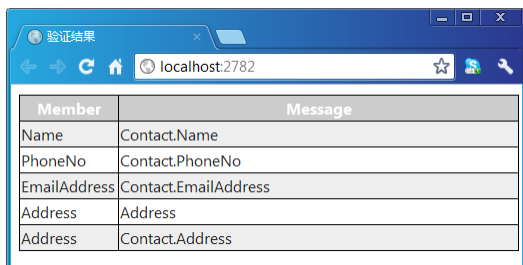
如下所示的是 Action 方法 Index 对应 View 的定义，可以看出这是一个 Model 类型为 `IEnumerable<ModelValidationResult>` 的强类型 View。在该 View 中我们将集合中的每一个 `ModelValidationResult` 对象的成员名称和错误消息通过表格的形式呈现出来。

```

@model IEnumerable<ModelValidationResult>
<html>
    <head>
        <title>验证结果</title>
    </head>
    <body>
        <table>
            <tr><th>Member</th><th>Message</th></tr>
            @foreach (ModelValidationResult result in Model)
            {
                string propertyName = string.IsNullOrEmpty(result.MemberName) ?
                    "N/A" : result.MemberName;
                <tr><td>@propertyName</td><td>@result.Message</td></tr>
            }
        </table>
    </body>
</html>

```

该程序运行后会在浏览器中呈现出如图 6-4 所示的输出结果，可以看出 `CompositeModelValidator` 对 `Contact` 对象实施验证得到的 5 个 `ModelValidationResult` 都来源于针对 4 个属性的验证，应用在 `Contact` 类型上的 `AlwaysFailsAttribute` 特性并没有参与验证。（S603）



Member	Message
Name	Contact.Name
PhoneNo	Contact.PhoneNo
EmailAddress	Contact.EmailAddress
Address	Address
Address	Contact.Address

图 6-4 CompositeModelValidator 的验证规则（1）

按照前面介绍的“针对容器对象本身的验证只有在所有属性通过验证的情况下才会进行”的原理，为了让 `Contact` 的四个属性通过验证，我们将应用在四个属性和 `Address` 类型上的 `AlwaysFailsAttribute` 特性注释掉，只保留应用在 `Contact` 类型和 `Address` 四个属性上的 `AlwaysFailsAttribute` 特性。再次运行我们的程序将会在浏览器中得到如图 6-5 所示的输出结果，不难看出输出的 `ModelValidationResult` 来源于应用于 `Contact` 类型上的 `AlwaysFailsAttribute` 特性。（S604）

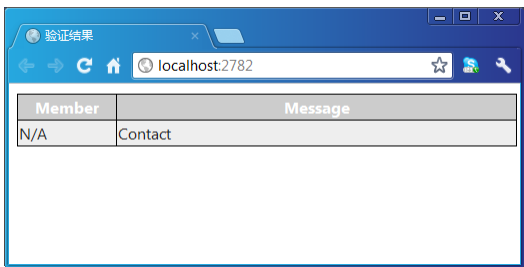


图 6-5 CompositeModelValidator 的验证规则（2）

6.2 Model 绑定与验证

Model 绑定解决了针对目标 Action 方法参数的绑定，而 Model 验证的目的在于对绑定的对象实施验证以确保输入数据的有效性，Model 验证是伴随着 Model 绑定进行的。在上面一节中我们详细地介绍了真正用于 Model 验证的 ModelValidator 以及相关的提供机制，接下来讨论在这个以 ModelValidator 为核心的 Model 验证系统中，针对通过 Model 绑定得到的数据对象的验证是如何实现的。

6.2.1 ModelState

通过第 5 章“Model 的绑定”的介绍我们知道，Controller 对象的 ViewData 包含一个元素类型为 ModelState 的集合，用于表示 Model 状态。除了在 Model 绑定过程通过 ValueProvider 提供的数据之外，提供数据的验证结果也保存其中。

```
[Serializable]
public class ModelState
{
    public ModelErrorCollection Errors { get; }
    public ValueProviderResult Value { get; set; }
}

[Serializable]
public class ModelErrorCollection : Collection<ModelError>
{
    public ModelErrorCollection();
    public void Add(Exception exception);
    public void Add(string errorMessage);
}

[Serializable]
public class ModelError
{
    public ModelError(Exception exception);
    public ModelError(string errorMessage);
    public ModelError(Exception exception, string errorMessage);

    public string ErrorMessage { get; }
    public Exception Exception { get; }
}
```


通过上面的代码片段所示, ModelState 具有 Value 和 Errors 两个核心属性,前者表示 ValueProvider 提供的 ValueProviderResult 对象,后者表示针对该数据对象的错误集合。Error 属性的类型为 System.Web.Mvc.ModelErrorCollection,这是一个元素类型为 System.Web.Mvc.ModelError 的集合,而一个 ModelError 对象通过错误消息和异常来描述错误。

实例演示: 验证 Model 绑定过程中对 ModelError 的设置 (S605)

Model 验证可以看成是 Model 绑定过程的一部分,它在参数列表创建过程中会对提供的数据实施验证,而表示验证结果的 ModelValidationResult 集合会以 ModelError 的形式写入当前 ViewData 的 ModelState 中。现在我们通过一个简单的实例来证实这一点,可以直接使用前面演示实例中创建的 Contact 作为验证类型,Contact 和 Address 类型和属性均应用了上面定义的 AlwaysFailsAttribute 特性。

我们定义了如下一个 HomeController,在无参 Action 方法 Index (针对 HTTP-GET 方法)中我们创建一个 Contact 对象并将其作为 Model 呈现在默认的 View 中。应用了 HttpPostAttribute 特性的 Index 方法具有一个类型为 Contact 的参数,此方法直接呈现默认的 View。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        Address address = new Address
        {
            Province = "江苏",
            City      = "苏州",
            District  = "工业园区",
            Street    = "星湖街 328 号"
        };
        Contact contact = new Contact
        {
            Name       = "张三",
            PhoneNo    = "123456789",
            EmailAddress = "zhangsan@gmail.com",
            Address     = address
        };
        return View(contact);
    }

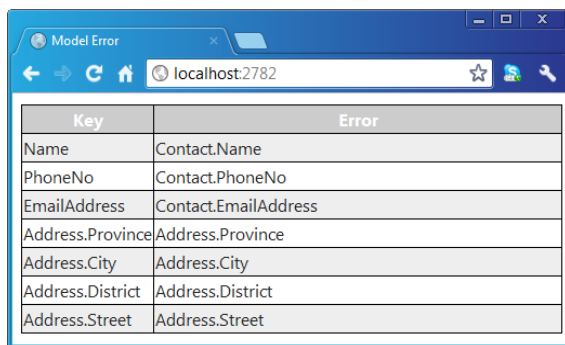
    [HttpPost]
    public ActionResult Index(Contact contact)
    {
        return View(contact);
    }
}
```

如下所示的是 Action 方法 Index 对应 View 的定义,可以看出这是一个 Model 类型为 Contact 的强类型 View。在该 View 中,如果当前不是针对 HTTP-POST 的请求,我们会将作为 Model 的 Contact 对象(连同 Address 属性)以编辑模式呈现在一个表单中,该表单具有一个提交按钮。对于 HTTP-POST 请求来说,我们将保存在当前 ViewData 中所有 ModelState

的 Key 以及该 ModelState 的错误消息以表格的方式呈现出来。

```
@model Contact
<html>
<head>
<title>Model Error</title>
</head>
<body>
@if (string.Compare(Request.HttpMethod, "POST", true) != 0)
{
    using(Html.BeginForm())
    {
        @Html.EditorForModel()
        @Html.EditorFor(m=>m.Address)
        <input type="submit" value="保存" />
    }
}
else
{
<table>
<tr><th>Key</th><th>Error</th></tr>
@foreach (string key in ViewData.ModelState.Keys)
{
    ModelError[] errors = ViewData.ModelState[key].Errors.ToArray();
    string firstError = errors.Any() ? errors[0].ErrorMessage : "N/A";
    <tr>
        <td rowspan="@errors.Length">@key</td><td>@firstError</td>
    </tr>
    for(int i=1; i<errors.Length; i++)
    {
        <tr><td>@errors[i].ErrorMessage</td></tr>
    }
}
</table>
}
</body>
</html>
```

该程序运行之后会先在浏览器中呈现一个编辑联系人的页面，我们直接点击“保存”按钮提交表单后会呈现出图 6-6 所示的输出结果，可以看到针对 Contact 和 Address 所有属性的验证错误信息被成功输出。



Key	Error
Name	Contact.Name
PhoneNo	Contact.PhoneNo
EmailAddress	Contact.EmailAddress
Address.Province	Address.Province
Address.City	Address.City
Address.District	Address.District
Address.Street	Address.Street

图 6-6 Model 验证过程中对 ModelState 的设置

对比前面演示 CompositeModelValidator 的实例，我们会发现另一个重要的现象，即 CompositeModelValidator 本身的验证过程不是递归进行的（针对 Contact 的验证过程中不会对 Address 的属性实施验证），但是伴随着 Model 绑定的整个验证过程却是递归进行的（Address 的属性的验证在针对 Contact 的验证过程中自动完成），我们将在后面详细讨论这个问题。

6.2.2 验证消息的呈现

Model 的验证是伴随着 Model 绑定完成的，当 ModelBinder 从请求中提取相应的数据为目标 Action 方法绑定参数值后，验证错误信息已经以 ModelState 的形式保存到相应的 ModelState 中。而 ModelState 列表属于当前 ViewData 的一部分，可以直接在 View 中被使用，这对错误信息在 View 中的呈现提供了可能。现在我们就来讨论验证错误信息在 View 中如何呈现。

ValidationMessage/ValidationMessageFor

验证消息在 View 中的呈现可以借助 HtmlHelper/HtmlHelper<TModel>来实现。如下面的代码所示，HtmlHelper 和 HtmlHelper<TModel> 分别提供了若干 ValidationMessage 和 ValidationMessageFor 扩展方法重载。

```
public static class ValidationExtensions
{
    //其他成员
    public static MvcHtmlString ValidationMessage(this HtmlHelper htmlHelper,
        string modelName);
    public static MvcHtmlString ValidationMessage(this HtmlHelper htmlHelper,
        string modelName, IDictionary<string, object> htmlAttributes);
    public static MvcHtmlString ValidationMessage(this HtmlHelper htmlHelper,
        string modelName, object htmlAttributes);
    public static MvcHtmlString ValidationMessage(this HtmlHelper htmlHelper,
        string modelName, string validationMessage);
    public static MvcHtmlString ValidationMessage(this HtmlHelper htmlHelper,
        string modelName, string validationMessage,
        IDictionary<string, object> htmlAttributes);
    public static MvcHtmlString ValidationMessage(this HtmlHelper htmlHelper,
        string modelName, string validationMessage, object htmlAttributes);

    public static MvcHtmlString ValidationMessageFor<TModel, TProperty>(
        this HtmlHelper<TModel> htmlHelper,
        Expression<Func<TModel, TProperty>> expression);
    public static MvcHtmlString ValidationMessageFor<TModel, TProperty>(
        this HtmlHelper<TModel> htmlHelper,
        Expression<Func<TModel, TProperty>> expression,
        string validationMessage);
    public static MvcHtmlString ValidationMessageFor<TModel, TProperty>(
        this HtmlHelper<TModel> htmlHelper,
        Expression<Func<TModel, TProperty>> expression, string validationMessage,
        IDictionary<string, object> htmlAttributes);
    public static MvcHtmlString ValidationMessageFor<TModel, TProperty>(
        this HtmlHelper<TModel> htmlHelper,
        Expression<Func<TModel, TProperty>> expression, string validationMessage,
        object htmlAttributes);
}
```

ViewData 的 ModelState 属性的类型并不是 ModelState，而是 ModelStateDictionary。HtmlHelper 的扩展方法 ValidationMessage 的参数 modelName 表示对应的 ModelState 在 ModelStateDictionary 中的 Key。如果针对这个 Key 找不到对应的 ModelState，或者对应的 ModelState 的 Errors 列表为空，意味着对应的数据成功通过验证，此时不会有任何 HTML 生成。

如果对应的 Errors 列表不为空，方法会生成一个元素来显示验证消息。如果通过 validationMessage 显示指定的验证消息，那么该消息将会直接作为该元素的内部文本，否则 Errors 列表中第一个非空消息将会作为验证消息。此外，当我们调用扩展方法 ValidationMessage 的时候还可以通过参数 htmlAttributes 为这个元素设置相应的 HTML 属性。ValidationMessageFor 与 ValidationMessage 不同之处在于它会通过指定的表达式来提取 ValidationMessage 方法中的参数 modelName。

现在我们对上面演示的实例（S605）略加改动来演示验证消息的呈现。如下面的代码片段所示，我们在应用了 HttpPostAttribute 特性的 Index 方法中将作为参数的 Contact 对象作为 Model 呈现在一个名为“ValidationMessage”的 View 中。

```
public class HomeController : Controller
{
    //其他成员
    [HttpPost]
    public ActionResult Index(Contact contact)
    {
        return View("ValidationMessage", contact);
    }
}
```

如下所示的是这个名为 ValidationMessage 的 View 的定义，这是一个 Model 类型为 Contact 的强类型 View，在该 View 中我们调用 HtmlHelper 的 ValidationMessage 扩展方法将所有的验证消息呈现出来。

```
@model Contact
<html>
<head>
<title>ValidationMessage</title>
<style type="text/css">
    .field-validation-error{color:Red}
</style>
</head>
<body>
<ul>
<li>@Html.ValidationMessage("Name")</li>
<li>@Html.ValidationMessage("PhoneNo")</li>
<li>@Html.ValidationMessage("EmailAddress")</li>
<li>@Html.ValidationMessage("Address.Province")</li>
<li>@Html.ValidationMessage("Address.City")</li>
<li>@Html.ValidationMessage("Address.District")</li>
<li>@Html.ValidationMessage("Address.Street")</li>
</ul>
</body>
</html>
```

运行该程序后，在联系人编辑页面中直接点击“保存”按钮，这个名为 ValidationMessage

的 View 会以如图 6-7 所示的效果呈现出来。(S606)

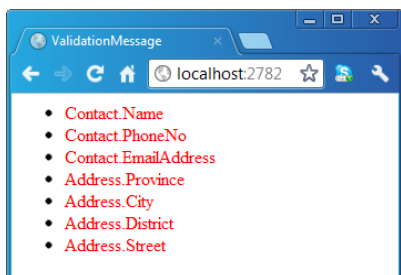


图 6-7 验证消息的呈现效果

在 ValidationMessage 中针对验证消息的呈现也可以按照如下的方式调用 HtmlHelper<TModel>的扩展方法 ValidationMessageFor 来实现。(S607)

```
<ul>
  <li>@Html.ValidationMessageFor(c=>c.Name)</li>
  <li>@Html.ValidationMessageFor(c=>c.PhoneNo)</li>
  <li>@Html.ValidationMessageFor(c=>c.EmailAddress)</li>
  <li>@Html.ValidationMessageFor(c=>c.Address.Province)</li>
  <li>@Html.ValidationMessageFor(c=>c.Address.City)</li>
  <li>@Html.ValidationMessageFor(c=>c.Address.District)</li>
  <li>@Html.ValidationMessageFor(c=>c.Address.Street)</li>
</ul>
```

通过这两个呈现出来的验证消息具有相同的显示效果，它们会生成如下所示的 HTML。可以看出呈现出来的验证显示体现为一个元素，样式类型（class="field-validation-error"）和客户端验证属性（data-valmsg-for="PhoneNo" data-valmsg-replace="true"）作了相应设置。

```
<ul>
  <li>
    <span class="field-validation-error" data-valmsg-for="Name"
      data-valmsg-replace="true">Contact.Name</span></li>
  <li>
    <span class="field-validation-error" data-valmsg-for="PhoneNo"
      data-valmsg-replace="true">Contact.PhoneNo</span>
  </li>
  <li>
    <span class="field-validation-error" data-valmsg-for="EmailAddress"
      data-valmsg-replace="true">Contact.EmailAddress</span>
  </li>
  <li>
    <span class="field-validation-error" data-valmsg-for="Address.Province"
      data-valmsg-replace="true">Address.Province</span>
  </li>
  <li>
    <span class="field-validation-error" data-valmsg-for="Address.City"
      data-valmsg-replace="true">Address.City</span>
  </li>
  <li>
    <span class="field-validation-error" data-valmsg-for="Address.District"
      data-valmsg-replace="true">Address.District</span>
  </li>
</ul>
```

```

</li>
<li>
    <span class="field-validation-error" data-valmsg-for="Address.Street"
        data-valmsg-replace="true">Address.Street</span>
</li>
</ul>

```

ValidationSummary

除了通过 `ValidationMessageFor` 与 `ValidationMessage` 这两个方法显示单条验证消息之外，我们还可以通过调用 `HtmlHelper` 的扩展方法 `ValidationSummary` 将所有的验证消息一并显示出来。如下面的代码片段所示，`HtmlHelper` 具有一系列 `ValidationSummary` 扩展方法重载，布尔类型的参数 `excludePropertyErrors` 表示是否需要排除基于属性的错误消息，而通过 `message` 参数可以为 `ValidationSummary` 指定一个作为标题的字符串。

```

public static class ValidationExtensions
{
    //其他成员
    public static MvcHtmlString ValidationSummary(this HtmlHelper htmlHelper);
    public static MvcHtmlString ValidationSummary(this HtmlHelper htmlHelper,
        bool excludePropertyErrors);
    public static MvcHtmlString ValidationSummary(this HtmlHelper htmlHelper,
        string message);
    public static MvcHtmlString ValidationSummary(this HtmlHelper htmlHelper,
        bool excludePropertyErrors, string message);
    public static MvcHtmlString ValidationSummary(this HtmlHelper htmlHelper,
        string message, IDictionary<string, object> htmlAttributes);
    public static MvcHtmlString ValidationSummary(this HtmlHelper htmlHelper,
        string message, object htmlAttributes);
    public static MvcHtmlString ValidationSummary(this HtmlHelper htmlHelper,
        bool excludePropertyErrors, string message,
        IDictionary<string, object> htmlAttributes);
    public static MvcHtmlString ValidationSummary(this HtmlHelper htmlHelper,
        bool excludePropertyErrors, string message, object htmlAttributes);
}

```

`ModelStateDictionary` 是一个 `Key` 和 `Value` 分别为字符串和 `ModelState` 的字典，并且允许一个空字符串作为其 `Key`。`ValidationSummary` 方法通过 `Key` 是否为空来判断 `ModelState` 是否针对一个属性。`ModelStateDictionary` 还定义了如下两个 `AddModelError` 方法重载使我们很容易地进行 `ModelError` 的设置。在该方法执行过程中，如果具有相同 `Key` 的 `ModelState` 对象存在，那么被添加的 `ModelError` 将会直接存放到它的 `Errors` 集合中，否则会添加到一个新创建的 `ModelState` 的 `Errors` 集合中。

```

[Serializable]
public class ModelStateDictionary : IDictionary<string, ModelState>,
    ICollection<KeyValuePair<string, ModelState>>,
    IEnumerable<KeyValuePair<string, ModelState>>, IEnumerable
{
    //其他成员
    public void AddModelError(string key, Exception exception);
    public void AddModelError(string key, string errorMessage);
}

```

我们在一个 ASP.NET MVC 应用中定义了如下一个默认的 HomeController。在默认的动作方法 Index 中添加了四个 ModelState 到当前的 ModelState 集合中,除了最后一个将一个空字符串作为 Key 之外,前三个均具有一个明确的 Key,最后我们直接将默认的 View 呈现出来。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ModelState.AddModelError("Name", "请输入姓名");
        ModelState.AddModelError("PhoneNo", "请输入电话号码");
        ModelState.AddModelError("EmailAddress", "请输入电子邮箱地址");

        ModelState.AddModelError("", "系统发生异常, 详细信息请与管理员联系");
        return View();
    }
}
```

如下所示的动作方法 Index 对应 View 的定义,在该 View 中我们两次调用 HtmlHelper 的 ValidationSummary 方法并且指定了 message 参数,ValidationSummary 方法的参数 excludePropertyErrors 在两次调用中分别设置为 False 和 True。

```
<html>
<head>
    <title>ValidationSummary</title>
    <style type="text/css">
        .validation-summary-errors{ color:Red}
        .validation-summary-errors span{ font-weight:bold}
    </style>
</head>
<body>
    @Html.ValidationSummary(false, "excludePropertyErrors: false")
    @Html.ValidationSummary(true, "excludePropertyErrors: true")
</body>
</html>
```

该程序运行之后会在浏览器中呈现如图 6-8 所示的效果,可以看到当 excludePropertyErrors 参数被设置为 True 的时候,ValidationSummary 中只会呈现出 Key 为空字符串的 ModelState 的错误消息。(S608)

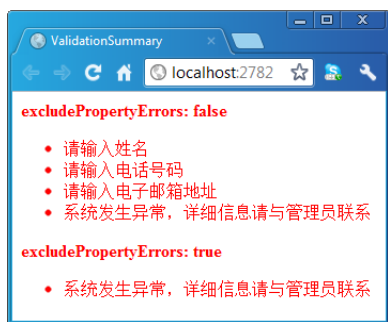


图 6-8 验证消息在 ValidationSummary 中的呈现效果

EditorForModel

在一个强类型 View 中, 当我们调用 `HtmlHelper<TModel>` 的扩展方法 `EditorForModel` 将整个 Model 对象以编辑模式呈现出来时, 如果某个属性对应的 `ModelState` 具有相应的错误 (通过 `Errors` 属性表示的 `ModelError` 集合不为空), 错误消息也会一并呈现出来。当然, 如果我们为 Model 类型定义了相应的模板就另当别论了。

我们同样可以通过一个简单的实例来演示错误消息在 `EditForModel` 方法中的呈现, 我们可以在一个 ASP.NET MVC 应用中定义了如下一个熟悉的 `Contact` 类型作为 View 的 Model。

```
public class Contact
{
    [DisplayName("姓名")]
    public string Name { get; set; }

    [DisplayName("电话号码")]
    public string PhoneNo { get; set; }

    [DisplayName("电子邮箱地址")]
    public string EmailAddress { get; set; }
}
```

然后创建一个具有如下定义的 `HomeController`。在 Action 方法 `Index` 中, 通过调用当前 `ModelState` 属性的 `AddModelError` 方法人为地添加三个错误消息, 对应的 `ModelState` 名称与作为 Model 的 `Contact` 类型的属性名称一致, 最后将创建的 `Contact` 对象在默认的 View 中呈现出来。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ModelState.AddModelError("Name", "请输入姓名");
        ModelState.AddModelError("PhoneNo", "请输入电话号码");
        ModelState.AddModelError("EmailAddress", "请输入电子邮箱地址");
        return View(new Contact());
    }
}
```

下面的代码片段代表了 Action 方法 `Index` 对应 View 的定义, 这是一个 Model 类型为 `Contact` 的强类型 View。在该 View 中我们仅仅简单地调用 `HtmlHelper<TModel>` 的扩展方法 `EditorForModel` 作为 Model 的 `Contact` 对象以编辑的模式呈现出来。

```
@model Contact
<html>
<head>
    <title>EditorForModel</title>
    <style type="text/css">
        .field-validation-error{color:Red}
    </style>
</head>
<body>
    @Html.EditorForModel()
</body>
</html>
```


当我们成功运行该程序的时候会在浏览器中呈现出如图 6-9 所示的效果，可以看到错误消息被显示在对应的文本框后面。（S609）

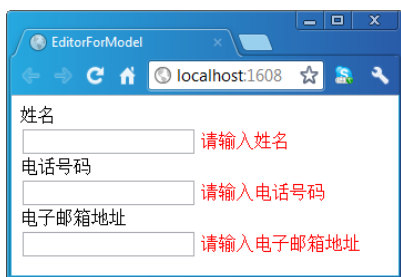


图 6-9 错误消息在 EditorForModel 方法中的呈现

6.2.3 Model 绑定中的验证

在前面我们不止一次地提到，Model 验证可以看成是 Model 绑定的一个中间环节，默认的情况下的 Model 绑定实现在 DefaultModelBinder 中。那么现在有这么一个问题，是 DefaultModelBinder 得到最终的参数对象后，再递交给 ModelValidator 实施验证呢，还是在实施 Model 绑定的过程中动态地调用 ModelValidator 对由 ValueProvider 提供的数据值实施验证？

实际上我们上面演示的两个实例已经回答了这个问题。通过上面演示的两个例子我们知道，CompositeModelValidator 这个默认 ModelValidator 在进行 Model 验证过程中并不是递归进行的（S603），但是从整个 Model 绑定过程来看，Model 验证却具有递归性，所以 Model 绑定和 Model 验证绝对不可能是先后的过程，唯一的可能是 DefaultModelBinder 在递归地进行 Model 绑定的过程中调用 ModelValidator 对提供的数据实施验证。

同样以针对 Contact 类型的 Model 绑定为例，当 DefaultModelBinder 通过 Model 得到一个被初始化的空 Contact 对象之后，会将描述 Contact 类型的 ModelMetadata 对象作为参数调用 ModelValidator 的静态方法 GetModelValidator，得到的 CompositeModelValidator 被用于对 Contact 对象实施验证。由于 CompositeModelValidator 的 Model 验证不具有递归性，所以只有应用在 Contact 四个属性（Name、PhoneNo、Email 和 Address）及其自身类型上的验证规则在本轮验证中有效。

由于 Contact 的 Address 属性是一个复杂类型，所以 DefaultModelBinder 在针对 Contact 类型的 Model 绑定过程中会递归地创建一个空 Address 对象作为 Contact 对象的 Address 属性。在完成对 Address 对象的绑定之后，又会调用 ModelValidator 的静态方法 GetModelValidator 根据描述 Address 类型的 ModelMetadata 得到一个 CompositeModelValidator，初始化后的 Address 对象被将交给它验证。

描述 Model 元数据的 ModelMetadata 具有一个树型层次化结构，我们的验证规则可以应用到每一个节点上。DefaultModelBinder 就是在递归地绑定复杂对象的过程中对绑定后的对

象实施验证,从而使各个层次上的验证得以实现。不过 `CompositeModelValidator` 只有在所有属性值都验证通过的情况下,才会使用应用在类型上的验证规则对数据对象实施验证,所以验证的结果也不能完全反映所有的验证规则。

实例演示: 模拟 Model 绑定中的验证 (S610)

在第 5 章“Model 的绑定”中,我们自定义了一个 `DefaultModelBinder` 实现了针对简单类型、复杂类型、数组、集合和字典的 Model 绑定。在本例中,我们在这个自定义的 `ModelBinder` 中引入 Model 验证部分。

通过前面的介绍我们知道,真正实施 Model 验证的是通过 `ModelValidator` 的静态方法 `GetModelValidator` 创建的 `CompositeModelValidator` 对象,那么我们按照其采用的 Model 验证逻辑自定义这么一个类型。如下面的代码片段所示,我们自定义的 `CompositeModelValidator` 直接继承自 `ModelValidator`。

```
public class CompositeModelValidator: ModelValidator
{
    public CompositeModelValidator(ModelMetadata metadata,
        ControllerContext controllerContext)
        : base(metadata, controllerContext)
    { }

    public override IEnumerable<ModelValidationResult> Validate(object container)
    {
        bool isPropertiesValid = true;
        foreach (ModelMetadata propertyMetadata in Metadata.Properties)
        {
            foreach (ModelValidator validator in
                propertyMetadata.GetValidators(this.ControllerContext))
            {
                IEnumerable<ModelValidationResult> results =
                    validator.Validate(propertyMetadata.Model);
                if (results.Any())
                {
                    isPropertiesValid = false;
                }
                foreach (ModelValidationResult result in results)
                {
                    yield return new ModelValidationResult
                    {
                        MemberName = DefaultModelBinder.CreateSubPropertyName(
                            propertyMetadata.PropertyName, result.MemberName),
                        Message = result.Message
                    };
                }
            }
        }

        if (isPropertiesValid)
        {
            foreach (ModelValidator validator in
                Metadata.GetValidators(this.ControllerContext))
            {

```

```

        IEnumerable<ModelValidationResult> results =
            validator.Validate(Metadata.Model);
        foreach (ModelValidationResult result in results)
        {
            yield return result;
        }
    }
}
}

```

定义在 `Validate` 的 `Model` 验证逻辑是这样的：先通过 `ModelMetadata` 属性获取当前的 `Model` 元数据，然后遍历所有描述属性的 `ModelMetadata`。对于每一个基于属性的 `ModelMetadata` 对象，我们通过调用其 `GetModelValidators` 方法得到应用在该属性上的 `ModelValidator` 列表。接下来调用列表中每一个 `ModelValidator` 的 `Validate` 方法对属性值进行验证，并根据返回值创建相应的 `ModelValidationResult` 对象，该对象被添加到最终返回的 `ModelValidationResult` 集合中。

只有在所有的属性通过验证的情况下，我们才根据当前 `ModelMetadata` 获取相应的 `ModelValidator` 列表对容器对象自身实施验证，验证的结果直接添加到最终返回的 `ModelValidationResult` 集合中。

需要注意一点的是，在进行针对属性的验证过程中，我们并没有直接使用返回 `ModelValidationResult` 对象，而是根据它创建了一个新的 `ModelValidationResult` 对象，该对象表示成员名称的 `MemberName` 属性被添加上了对应属性名前缀。用于计算成员名称的静态方法 `CreateSubPropertyName` 定义在 `DefaultModelBinder` 中，具体的定义如下。

```

public class DefaultModelBinder : IModelBinder
{
    //其他成员
    internal static string CreateSubPropertyName(string prefix,
        string propertyName)
    {
        prefix = prefix ?? "";
        propertyName = propertyName ?? "";
        return (prefix + "." + propertyName).Trim('.');
    }
}

```

在自定义的 `DefaultModelBinder` 中，我们定义单独的方法来完成针对简单类型、复杂类型、数组、集合和字典的 `Model` 绑定，而只需要在进行针对复杂类型的 `Model` 绑定过程中利用 `CompositeModelValidator` 进行 `Model` 验证。`CompositeModelValidator` 能够完成针对容器对象所有属性及其自身的验证，虽然它不递归地去验证复杂类型属性的数据成员，但是由于 `Model` 绑定本身是一个递归的过程，所以从整个流程上看 `Model` 验证是递归进行的。

针对复杂数据类型 `Model` 验证实现在 `DefaultModelBinder` 的 `GetComplexModel` 方法中。如下面的代码片段所示，我们在目标容器对象生成之后利用创建的 `CompositeModelValidator` 对象对它进行验证。对于返回的每一个 `ModelValidationResult`，我们将其错误消息添加到相应的 `ModelState` 之中。

```

public class DefaultModelBinder : IModelBinder
{
    //其他成员
    protected virtual object GetComplexModel(
        ControllerContext controllerContext, Type modelType,
        IValueProvider valueProvider, string prefix)
    {
        object model = CreateModel(modelType);
        foreach (PropertyDescriptor property in
            TypeDescriptor.GetProperties(modelType))
        {
            if (property.IsReadOnly)
            {
                continue;
            }
            string key = string.IsNullOrEmpty(prefix) ? property.Name :
                prefix + "." + property.Name;
            property.SetValue(model, GetModel(controllerContext,
                property.PropertyType, valueProvider, key));
        }

        //Model 验证
        ModelMetadata metadata = ModelMetadataProviders.Current
            .GetMetadataForType(() => model, modelType);
        CompositeModelValidator validator =
            new CompositeModelValidator(metadata, controllerContext);
        foreach (ModelValidationResult result in validator.Validate(model))
        {
            string key = CreateSubPropertyName(prefix, result.MemberName);
            controllerContext.Controller.ViewData.ModelState.AddModelError(key,
                result.Message);
        }
        return model;
    }
}

```

在第5章“Model 的绑定”中已经验证过了自定义 `DefaultModelBinder` 的 Model 绑定功能，现在通过一个简单的实例来验证刚刚增加的 Model 验证功能。我们直接使用上面实例中定义的 `Contact` 类型，并且在 `Contact` 和 `Address` 类型和属性上应用自定义的 `AlwaysFailsAttribute` 特性。接下来我们定义了如下一个 `HomeController`，针对 HTTP-GET 的 Action 方法直接将一个空 `Contact` 对象呈现在默认的 View 中，而应用了 `HttpPostAttribute` 特性的 `Index` 方法具有一个 `Contact` 类型的参数，该参数上应用了 `ModelBinderAttribute` 特性将我们自定义的 `DefaultModelBinder` 用于该参数的绑定。

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View(new Contact());
    }

    [HttpPost]
    public ActionResult Index(
        [ModelBinder(typeof(DefaultModelBinder))]

```

```

        Contact contact)
    {
        return View(contact);
    }
}

```

如下所示的是 Action 方法 Index 对应 View 的定义，这是一个 Model 类型为 Contact 的强类型 View。在该 View 中我们将作为 Model 的 Contact 对象的所有属性（包括 Address 的所有属性）以编辑模式呈现在一个表单之中，该表单具有一个用于提交的“保存”按钮。

```

@model Contact
<html>
<head>
    <title>Model 绑定中的验证</title>
    <style type="text/css">
        .field-validation-error{color:Red}
    </style>
</head>
<body>
    @using (Html.BeginForm())
    {
        @Html.EditorForModel()
        @Html.EditorFor(m=>m.Address)
        <input type="submit" value="保存" />
    }
</body>
</html>

```

该程序运行之后会现在浏览器中呈现一个“编辑联系人信息”的页面，我们直接点击“保存”按钮会呈现出如图 6-10 所示的输出结果，文本框右侧显示的文本正是应用在 Contact 和 Address 相应属性上的 AlwaysFailsAttribute 特性定义的错误消息。

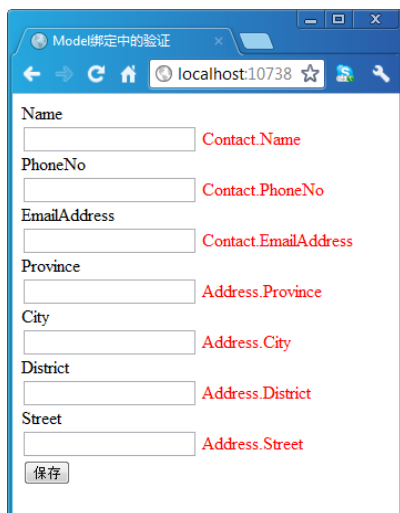


图 6-10 实现在自定义 DefaultModelBinder 中的 Model 验证

6.3 基于数据注解特性的 Model 验证

通过前面的介绍我们知道，`ModelValidatorProviders` 的静态只读属性 `Providers` 维护着一个全局的 `ModelValidatorProvider` 列表，最终用于 Model 验证的 `ModelValidator` 都是通过这些 `ModelValidatorProvider` 来提供的。对于该列表默认包含的三种 `ModelValidatorProvider` 来说，`DataAnnotationsModelValidatorProvider` 无疑是最重要的，ASP.NET MVC 默认提供的基于验证特性的声明式 Model 验证就是通过它提供的 `DataAnnotationsModelValidator` 来实现的。

6.3.1 ValidationAttribute 特性

与通过数据注解特性定义 `ModelMetadata` 类似，我们可以在数据类型及其属性上应用相应的验证特性来定义相应的验证规则，所有的验证特性都直接或者间接继承自具有如下定义的抽象类型 `System.ComponentModel.DataAnnotations.ValidationAttribute`。

```
public abstract class ValidationAttribute : Attribute
{
    public string      ErrorMessage { get; set; }
    public string      ErrorMessageResourceName { get; set; }
    public Type        ErrorMessageResourceType { get; set; }
    protected string  ErrorMessageString {get;}

    public virtual string FormatErrorMessage(string name);
    public virtual bool IsValid(object value);
    protected virtual ValidationResult IsValid(object value,
        ValidationContext validationContext)
    public void Validate(object value, string name);
    public ValidationResult GetValidationResult(object value,
        ValidationContext validationContext);
}
```

如上面的代码片段所示，`ValidationAttribute` 具有一个字符串类型的 `ErrorMessage` 属性用于指定错误消息。出于对本地化或者对错误消息单独维护的需要，可以采用资源文件的方式来维护错误消息，在这种情况下只需要通过 `ErrorMessageResourceName` 和 `ErrorMessageResourceType` 这两个属性指定错误消息所在资源项的名称和类型即可。如果我们通过 `ErrorMessage` 属性指定一个字符串作为验证错误消息，又通过 `ErrorMessageResourceName/ErrorMessageResourceType` 属性指定了错误消息资源项对应的名称和类型，后者具有更高的优先级。`ValidationAttribute` 具有一个受保护的只读属性 `ErrorMessageString` 用于返回最终的错误消息文本。

对于错误消息的定义，我们可以指定完整的消息内容，比如“年龄必须在 18 至 25 之间”。但是对于像资源文件这种对错误消息进行独立维护的情况，为了让定义的资源文本能够最大限度地被重用，我们倾向于定义一个包含占位符的文本模板，比如“{DisplayName}必须在 {LowerBound} 和 {UpperBound} 之间”，这样消息适用于所有基于数值范围的验证。模板中的占位符可以在虚方法 `FormatErrorMessage` 中进行替换，该方法中的参数 `name` 实际上代表的是对应的显示名称，即 `ModelMetadata` 的 `DisplayName` 属性。

FormatErrorMessage 方法在 ValidationAttribute 中仅仅是按照如下的方式调用 String 的静态方法 Format 将参数 name 作为替换占位符的参数,所以在默认的情况下我们定义错误消息模板只允许包含唯一一个针对显示名称的占位符“{0}”。如果具有额外的占位符,或者需要采用非序号(“{0}”)占位符定义方式(比如采用类似于“{DisplayName}”这种基于文字的占位符更具可读性),只需要重写 FormatErrorMessage 方法即可。

```
public abstract class ValidationAttribute : Attribute
{
    //其他成员
    public virtual string FormatErrorMessage(string name)
    {
        return string.Format(CultureInfo.CurrentCulture,
            ErrorMessageString, new object[] { name });
    }
}
```

当我们通过继承 ValidationAttribute 创建自己的验证特性的时候,可以通过重写任意一个 IsValid 方法来定义验证逻辑。之所以能够通过重写任意一个 IsValid 方法来实现自定义验证,原因在于定义在 ValidationAttribute 的这两个 IsValid 方法之间存在相互调用的关系。很显然,这种相互调用必然造成“死循环”,所以我们需要重写至少其中一个方法来避免“死循环”的发生。这里的“死循环”被加上引号,是因为 ValidationAttribute 在内部做了处理,当这种情况出现的时候会抛出一个 NotImplementedException 异常。

```
//调用公有 IsValid 方法
public class ValidatorAttribute : ValidationAttribute
{
    static void Main()
    {
        ValidatorAttribute validator = new ValidatorAttribute();
        validator.IsValid(new object());
    }
}

//调用受保护 IsValid 方法
public class ValidatorAttribute : ValidationAttribute
{
    static void Main()
    {
        ValidatorAttribute validator = new ValidatorAttribute();
        validator.IsValid(new object(), null);
    }
}
```

我们通过一个简单的实例来演示自定义 ValidationAttribute 对两个 IsValid 方法进行重写的必要性。在一个控制台应用中我们分别编写了如上两段程序,通过继承自 ValidationAttribute 自定义的 ValidatorAttribute 没有重写任何一个 IsValid 方法。当我们在 Debug 模式下分别运行这两段程序的时候,都会抛出如图 6-11 所示的 NotImplementedException 异常,提示“此类尚未实现 IsValid(object value)。首选入口点是 GetValidationResult(), 并且类应重写 IsValid(object value, ValidationContext context)。”

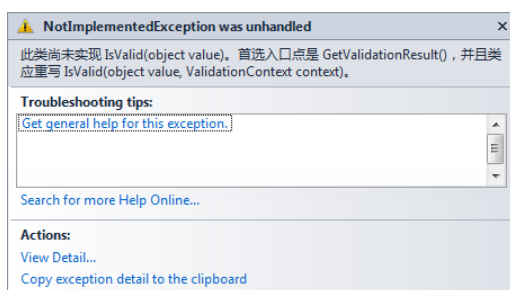


图 6-11 没有在自定义 ValidationAttribute 中重写 IsValid 方法导致的异常

受保护的 IsValid 方法中除了包含一个表示被验证对象的参数 value，还有一个类型为 ValidationContext 的参数。顾名思义，ValidationContext 旨在为当前的验证维护相应的上下文，这些信息包括通过 ObjectInstance 和 ObjectType 属性表示的验证对象及其类型，以及通过 MemberName 和 DisplayName 属性表示的成员名称（一般指属性名称）和显示名称。

```
public sealed class ValidationContext
{
    //其他成员
    public ValidationContext(object instance);
    public ValidationContext(object instance, IDictionary<object, object> items);

    public string      DisplayName { get; set; }
    public string      MemberName { get; set; }
    public object      ObjectInstance { get; }
    public Type        ObjectType { get; }
}
```

该 IsValid 方法返回值类型为 System.ComponentModel.DataAnnotations.ValidationResult。如下面的代码片段所示，它与作为 ModelValidator 验证结果的 ModelValidationResult 类型具有类似的定义，它依然是错误消息和成员名称的组合。不过 ModelValidationResult 对应某个单一的成员名称，而 ValidationResult 包含一组相关成员名称的列表。

```
public class ValidationResult
{
    //其他成员
    public ValidationResult(string errorMessage);
    public ValidationResult(string errorMessage,
        IEnumerable<string> memberNames);

    public string      ErrorMessage { get; set; }
    public IEnumerable<string> MemberNames { get; }
}
```

定义在 ValidationAttribute 中的 IsValid 方法在验证失败的情况下会返回一个 ValidationResult 对象，如果指定的 ValidationContext 不为 Null，那么其 MemberName 属性表示的成员名称将会包含在该 ValidationResult 对象的 MemberNames 列表中。ValidationContext

的 `DisplayName` 属性将会作为调用 `FormatErrorMessage` 的参数，而得到的格式化错误消息将会作为 `ValidationResult` 的 `ErrorMessage` 属性。如果成功通过验证，则直接返回 `Null`。

我们可以通过调用 `ValidationAttribute` 的方法 `GetValidationResult` 对指定的对象实施验证并得到以 `ValidationResult` 对象形式返回的验证结果，得到的 `ValidationResult` 对象实际上就是调用受保护 `IsValid` 方法的返回值。也可以调用 `Validate` 方法验证某个指定的对象，该方法在验证失败的情况下会直接抛出一个 `ValidationException` 异常，而通过调用 `FormatErrorMessage` 方法（将参数 `name` 表示的字符串作为参数）格式化后的错误消息将会作为该异常的消息。

在 `System.ComponentModel.DataAnnotations` 命名空间下定义了一系列具体的验证特性，它们大都直接应用在自定义数据类型的某个属性上目标数据成员实施验证。这些预定义验证特性不是本章论述的重点，所以在这里只是对它们作一个概括性的介绍。

- **RequiredAttribute**：用于验证必需数据成员。
- **RangeAttribute**：用于验证数据成员的值是否在指定的范围之内。
- **StringLengthAttribute**：用于字符串验证数据成员的长度是否在指定的范围之内。
- **MaxLengthAttribute/MinLengthAttribute**：用于验证字符/数组字典的数据成员长度是否小于/大于指定的上/下限。
- **RegularExpressionAttribute**：用于验证字符串数据成员的格式是否与指定的正则表达式相匹配。
- **CompareAttribute**：用于验证数据成员的值是否与另一个成员一致，在用户注册场景中可以用于确认两次输入密码的一致性。
- **CustomValidationAttribute**：指定一个用于验证目标成员的验证类型和验证方法。

应用 `ValidationAttribute` 特性的唯一性

对于上面列出的这些预定义 `ValidationAttribute`，它们都具有一个相同的特征，那就是在同一个目标元素中只能应用一次，这可以通过应用在它们上面的 `AttributeUsageAttribute` 特性的定义看出来。以如下所示的 `RequiredAttribute` 特性为例，应用在该类型上的 `AttributeUsageAttribute` 特性的 `AllowMultiple` 属性被设置为 `False`。

```
[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Field |
    AttributeTargets.Property, AllowMultiple=false)]
public class RequiredAttribute : ValidationAttribute
{
    //省略成员
}
```

但是是否意味着如果我们在自定义 `ValidationAttribute` 的时候将 `AttributeUsageAttribute` 特性的 `AllowMultiple` 设置为 `True`，它们就可以被多次应用到同一个属性或者类型上了呢？

我们不妨通过实例演示的方式来说明这个问题。

我们知道 `RangeAttribute` 可以帮助我们验证目标字段值的范围，但是有时候我们需要进行“条件性范围验证”。举个例子，我们现在对某个员工的薪水进行验证，但是不同级别的员工的薪水范围是不同的，为此我们创建了一个名为 `RangeIfAttribute` 的验证特性帮助我们进行针对不同级别的薪水范围验证。如下面的代码片段所示，我们将三个 `RangeIfAttribute` 特性应用到了表示薪水的 `Salary` 属性上，分别针对三个级别（G7、G8 和 G9）的薪水范围作了相应的设定。

```
public class Employee
{
    public string Name { get; set; }
    public string Grade { get; set; }

    [RangeIf("Grade", "G7", 2000, 3000)]
    [RangeIf("Grade", "G8", 3000, 4000)]
    [RangeIf("Grade", "G9", 4000, 5000)]
    public decimal Salary { get; set; }
}
```

如下面的代码片段所示，`RangeIfAttribute` 直接继承自 `RangeAttribute`。`RangeIfAttribute` 根据被验证容器对象的另一个属性值来决定是否对当前属性实施验证，属性 `Property` 和 `Value` 就分别代表这个属性和与之匹配的值。在重写的 `IsValid` 方法中，我们通过反射获取到了容器对象用于匹配的属性值，如果该值与 `Value` 属性值相匹配，则调用基类同名方法对指定对象进行验证，否则直接返回 `ValidationResult.Success (Null)`。应用在 `RangeIfAttribute` 上的 `AttributeUsageAttribute` 特性的 `AllowMultiple` 被设置为 `True`。

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = true)]
public class RangeIfAttribute : RangeAttribute
{
    public string Property { get; set; }
    public string Value { get; set; }

    public RangeIfAttribute(string property, string value, double minimum,
        double maximum)
        : base(minimum, maximum)
    {
        this.Property = property;
        this.Value = value ?? "";
    }

    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        PropertyInfo property =
            validationContext.ObjectType.GetProperty(this.Property);
        object propertyValue =
            property.GetValue(validationContext.ObjectInstance, null);
        propertyValue = propertyValue ?? "";
        if (propertyValue.ToString() != this.Value)
        {
            return ValidationResult.Success;
        }
    }
}
```

```

    }
    return base.IsValid(value, validationContext);
}
}

```

那么这样一个 `RangeIfAttribute` 特性真的能够按照我们期望的方式进行验证吗？为此我们在创建的空 ASP.NET MVC 应用中定义了如下一个 `HomeController`。在 Action 方法 `Index` 中创建了用于描述 `Employee` 的 `Salary` 属性 `ModelMetadata` 对象，并通过调用其 `GetValidators` 方法得到针对该属性的所有 `ModelValidator` 列表，最终将这个 `ModelValidator` 列表转化为数组作为 `Model` 呈现在对应的 `View` 中。

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        ModelMetadata employeeMetadata = ModelMetadataProviders.Current
            .GetMetadataForType(() => new Employee(), typeof(Employee));
        ModelMetadata salaryMetadata = employeeMetadata.Properties
            .FirstOrDefault(p => p.PropertyName == "Salary");
        IEnumerable<ModelValidator> validators = salaryMetadata
            .GetValidators(ControllerContext);
        return View(validators.ToArray());
    }
}

```

如下所示的 Action 方法 `Index` 对应 `View` 的定义，这是一个 `Model` 类型为 `ModelValidator` 数组的强类型 `View`。在该 `View` 中，我们将所有 `ModelValidator` 对象的类型名称通过表格的形式呈现出来。

```

@model ModelValidator[]
<html>
<head>
    <title>ModelValidators</title>
</head>
<body>
    <table>
        @for(int i= 0; i<Model.Length; i++)
        {
            <tr><td>@(i+1)</td><td>@Model[i].GetType().Name</td></tr>
        }
    </table>
</body>
</html>

```

该程序运行之后会在浏览器中呈现出如图 6-12 所示输出结果。由于 `Employee` 的 `Salary` 属性类型为非空值类型，所以会自动添加一个 `RequiredAttributeAdapter` 来进行必要性验证，另一个用于数值验证的 `NumericModelValidator` 也是源于 `Salary` 属性的类型。实际上只有第一个 `DataAnnotationsModelValidator` 是针对应用在 `Salary` 属性上的 `RangeIfAttribute` 特性而创建的，换句话说，应用在同一属性上的三个 `RangeIfAttribute` 特性只有一个是有效的，具有原因何在呢？（S611）

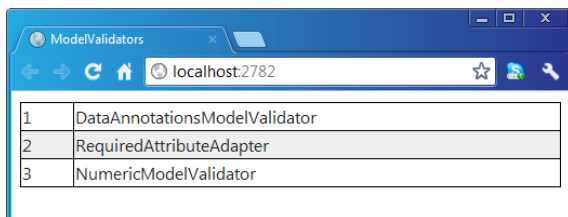


图 6-12 应用了多个同类验证特性的属性具有的 ModelValidator (1)

我们知道 Attribute 具有一个 object 类型的 TypeId 属性，默认返回代表自身类型的 Type 对象。ASP.NET MVC 在根据 ValidationAttribute 特性创建相应的 DataAnnotationsModelValidator 对象的时候会根据该 TypeId 属性值进行分组，同一组的 ValidationAttribute 只会选择第一个。这就意味着对于多个应用到相同目标元素同类 ValidationAttribute，有且只有一个是有有效的。

那么如何解决这个问题呢？其实很简单，既然 Model 验证系统会根据 TypeId 对所有验证特性进行筛选，我们只需要通过重写 TypeId 属性使每个 ValidationAttribute 具有不同的属性值就可以了，为此我们按照如下的方式在 RangeIfAttribute 中重写了 TypeId 属性。

```
[AttributeUsage( AttributeTargets.Field| AttributeTargets.Property,
    AllowMultiple = true)]
public class RangeIfAttribute: RangeAttribute
{
    //其他成员
    private object typeid;
    public override object TypeId
    {
        get{ return typeid?? (typeid= new object());}
    }
}
```

再次运行程序后将会在浏览器中得到如图 6-13 所示的输出结果，可以看到针对三个 RangeIfAttribute 特性的三个 DataAnnotationsModelValidator 被创建出来了。顺便说一下，通过重写 TypeId 而使多个 ValidationAttribute 可以同时应用到相同的属性或者类型的解决方案并不适合客户端验证，因为这会导致多组相同的验证规则被生成，而这是不允许的。(S612)

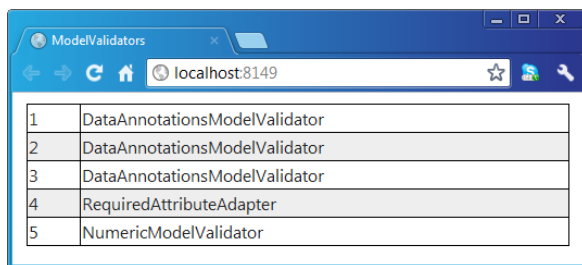


图 6-13 应用了多个同类验证特性的属性具有的 ModelValidator (2)

6.3.2 DataAnnotationsModelValidator

ModelValidator 是真正用于 **Model** 验证的组件，应用在数据类型及其属性上的验证特性最终被转换成相应的 **ModelValidator** 参与到针对目标数据的验证中，这个 **ModelValidator** 类型就是具有如下定义的 **DataAnnotationsModelValidator**，它的只读属性 **Attribute** 返回的就是对应的验证特性。

```
public class DataAnnotationsModelValidator : ModelValidator
{
    public DataAnnotationsModelValidator(ModelMetadata metadata,
        ControllerContext context, ValidationAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();

    public override IEnumerable<ModelValidationResult> Validate(object container)
    {
        ValidationContext validationContext = new ValidationContext(
            container ?? this.Metadata.Model, null, null)
        {
            DisplayName = this.Metadata.GetDisplayName()
        };
        ValidationResult validationResult = this.Attribute.GetValidationResult(
            this.Metadata.Model, validationContext);
        if (validationResult != ValidationResult.Success)
        {
            ModelValidationResult iteratorVariable2 = new ModelValidationResult
            {
                Message = validationResult.ErrorMessage
            };
            yield return iteratorVariable2;
        }
        else
        {
            yield break;
        }
    }
    protected ValidationAttribute Attribute { get; }
    protected string ErrorMessage { get; }
    public override bool IsRequired { get; }
}
```

上面的代码片段给出了用于实施验证的核心方法 **Validate** 的完整定义。该方法首先针对被验证容器对象创建出表示验证上下文的 **ValidationContext** 对象，并采用 **ModelMetadata** 的 **DisplayName** 属性作为该上下文的显示名称。真正的验证工作通过调用封装的验证特性的 **GetValidationResult** 方法来完成，如果该方法返回值不为 **Null** (**ValidationResult.Success**)，则将返回的 **ValidationResult** 转换成 **ModelValidationResult** 对象并添加到最终返回的 **ModelValidationResult** 集合中。

顺便再说说定义在 **DataAnnotationsModelValidator** 中的另外两个受保护只读属性的逻辑。用于返回错误消息的 **ErrorMessage** 属性来源于对验证特性的 **FormatErrorMessage** 方法的调用，而指定的参数就是当前 **ModelMetadata** 的 **DisplayName** 属性。由于只有

RequiredAttribute 特性才会对被验证数据实施必要性验证，所以只有被封装的 ValidationAttribute 为 RequiredAttribute 时其 IsRequired 属性才返回 True。

除了 DataAnnotationsModelValidator，ASP.NET MVC 还定义了一个具有如下定义的泛型的 System.Web.Mvc.DataAnnotationsModelValidator<TAttribute>，它是 DataAnnotationsModelValidator 的子类，泛型参数表示被封装的验证特性的类型。

```
public class DataAnnotationsModelValidator<TAttribute> :
    DataAnnotationsModelValidator where TAttribute: ValidationAttribute
{
    public DataAnnotationsModelValidator(ModelMetadata metadata,
        ModelBindingExecutionContext context, TAttribute attribute);
    protected TAttribute Attribute { get; }
}
```

ASP.NET MVC 为 4 个常用的验证特性（RequiredAttribute、RangeAttribute、RegularExpressionAttribute 和 StringLengthAttribute）定义了相应的适配类型。如下面的代码片段所示，它们都是泛型的 DataAnnotationsModelValidator<TAttribute> 的子类。当我们将这些 ValidationAttribute 应用到 Model 类型时，真正用于 Model 验证的实际上就是这些作为适配器的 ModelValidator。

```
public class RequiredAttributeAdapter :
    DataAnnotationsModelValidator<RequiredAttribute>
{
    public RequiredAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, RequiredAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

public class RangeAttributeAdapter :
    DataAnnotationsModelValidator<RangeAttribute>
{
    public RangeAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, RangeAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

public class RegularExpressionAttributeAdapter :
    DataAnnotationsModelValidator<RegularExpressionAttribute>
{
    public RegularExpressionAttributeAdapter(ModelMetadata metadata,
        ControllerContext context, RegularExpressionAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

public class StringLengthAttributeAdapter :
    DataAnnotationsModelValidator<StringLengthAttribute>
{
    public StringLengthAttributeAdapter(ModelMetadata metadata,
```

```

        ControllerContext context, StringLengthAttribute attribute);
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules();
}

```

6.3.3 DataAnnotationsModelValidatorProvider

`DataAnnotationsModelValidator` 最终是通过对应的 `DataAnnotationsModelValidatorProvider` 创建的。通过前面的介绍我们知道它是 `AssociatedValidatorProvider` 的子类，后者在用于获取 `ModelValidator` 的 `GetValidators` 方法中已经根据指定的 `ModelMetadata` 将所有特性提取出来，`DataAnnotationsModelValidatorProvider` 只需要从中筛选出继承自 `ValidationAttribute` 的验证特性并创建对应的 `DataAnnotationsModelValidator` 就可以了。

我们现在结合 `DataAnnotationsModelValidatorProvider` 的相关定义来讨论一下具体的 `ModelValidator` 提供机制。如下面的代码片段所示，它具有两个静态的字段 `AttributeFactories` 和 `DefaultAttributeFactory`，后者是一个类型为 `DataAnnotationsModelValidationFactory` 的委托，前者是以此委托为 Value 以 Type 对象为 Key 的字典。

```

public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    //其他成员
    internal static readonly Dictionary<Type,
        DataAnnotationsModelValidationFactory> AttributeFactories;
    internal static DataAnnotationsModelValidationFactory
        DefaultAttributeFactory;

    internal static DataAnnotationsValidatableObjectAdapterFactory
        DefaultValidatableFactory;
    internal static readonly Dictionary<Type,
        DataAnnotationsValidatableObjectAdapterFactory> ValidatableFactories;

    protected override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, HttpContext actionContext,
        IEnumerable<Attribute> attributes);
}

public delegate ModelValidator
    DataAnnotationsModelValidationFactory(ModelMetadata metadata,
        ControllerContext context, ValidationAttribute attribute);

public delegate ModelValidator
    DataAnnotationsValidatableObjectAdapterFactory(ModelMetadata metadata,
        ControllerContext context);

```

委托 `DataAnnotationsModelValidationFactory` 根据 `ModelMetadata`、`ControllerContext` 和 `ValidationAttribute` 创建一个 `ModelValidator` 对象。字段 `AttributeFactories` 表示的字典将验证特性的类型作为 Key，换句话说它维护一个 `ValidationAttribute` 特性类型和对应 `ModelValidator` 工厂的匹配关系。

在重写的 `GetValidators` 方法中，针对提供的每一个 `ValidationAttribute` 特性，它先根据

其类型从 `AttributeFactories` 字典中获取一个对应的 `DataAnnotationsModelValidationFactory` 委托。如果该委托对象存在，则用它来创建相应的 `ModelValidator` 对象，否则就采用字段 `DefaultAttributeFactory` 表示的 `DataAnnotationsModelValidationFactory` 委托来进行 `ModelValidator` 的创建。

除了 `AttributeFactories` 和 `DefaultAttributeFactory`, `DataAnnotationsModelValidatorProvider` 还具有 `DefaultValidatableFactory` 和 `ValidatableFactories` 两个静态字段，它们用于针对可验证对象（实现了 `IValidatableObject` 接口）的 `ModelValidator` 创建。字段 `DefaultValidatableFactory` 的类型是另外一个名为 `DataAnnotationsValidatableObjectAdapterFactory` 的委托，该委托根据 `ModelMetadata` 和 `ControllerContext` 创建相应的 `ModelValidator`。字段 `ValidatableFactories` 是一个以此委托为 Value、以 Type 对象为 Key 的字典。

当 `DataAnnotationsModelValidatorProvider` 完成了针对基于验证特性的 `ModelValidator` 的创建之后，如果被验证数据类型实现了 `IValidatableObject` 接口，它会先从静态字段 `ValidatableFactories` 中根据此类型获取一个对应的 `DataAnnotationsValidatableObjectAdapterFactory` 委托。如果匹配的委托对象存在，则用其进行 `ModelValidator` 的创建，否则采用字段 `DefaultValidatableFactory` 表示的默认工厂来创建相应的 `ModelValidator` 对象。

在 `DataAnnotationsModelValidatorProvider` 类型被加载的时候，上述的四个字段会被初始化。从如下的代码可以看出，一般的验证特性的 `ModelValidator` 是一个 `DataAnnotationsModelValidator` 对象（对应 `DefaultAttributeFactory` 字段）。`RangeAttribute`、`RegularExpressionAttribute`、`RequiredAttribute` 和 `StringLengthAttribute` 这四种验证特性，对应的 `ModelValidator` 是它们对应的适配器。对于可验证对象来说，默认情况下提供的 `ModelValidator` 列表中还包含一个 `ValidatableObjectAdapter` 对象。

```
public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    //其他成员
    static DataAnnotationsModelValidatorProvider()
    {
        //1.DefaultAttributeFactory
        DefaultAttributeFactory = (metadata, context, attribute)
            => new DataAnnotationsModelValidator(metadata, context, attribute);

        //2.AttributeFactories
        Dictionary<Type, DataAnnotationsModelValidationFactory> dictionary =
            new Dictionary<Type, DataAnnotationsModelValidationFactory>();
        dictionary.Add(typeof(RangeAttribute), (metadata, context, attribute)
            => new RangeAttributeAdapter(metadata, context,
                (RangeAttribute)attribute));
        dictionary.Add(typeof(RegularExpressionAttribute),
            (metadata, context, attribute) =>
                new RegularExpressionAttributeAdapter(metadata, context,
                    (RegularExpressionAttribute)attribute));
        dictionary.Add(typeof(RequiredAttribute), (metadata, context, attribute)
            => new RequiredAttributeAdapter(metadata, context,
                (RequiredAttribute)attribute));
    }
}
```



```

        dictionary.Add(typeof(StringLengthAttribute), (metadata, context,
            attribute) => new StringLengthAttributeAdapter(metadata, context,
                (StringLengthAttribute)attribute));
        AttributeFactories = dictionary;

        //3.DefaultValidatableFactory
        DefaultValidatableFactory = (metadata, context) =>
            new ValidatableObjectAdapter(metadata, context);

        //4.ValidatableFactories
        ValidatableFactories =
            new Dictionary<Type, DataAnnotationsValidatableObjectAdapterFactory>();
    }
}

```

DataAnnotationsModelValidatorProvider 四个基于委托的静态字段体现了采用的 **ModelValidator** 提供机制。由于它们都是内部字段，我们不能直接对其进行操作，但是如下所示的一系列静态方法定义在 **DataAnnotationsModelValidatorProvider** 之中，我们可以按照具体的需要调用它们对默认的 **ModelValidator** 进行注册。

```

public class DataAnnotationsModelValidatorProvider :
    AssociatedValidatorProvider
{
    //其他成员
    public static void RegisterAdapter(Type attributeType, Type adapterType);
    public static void RegisterAdapterFactory(Type attributeType,
        DataAnnotationsModelValidationFactory factory);
    public static void RegisterDefaultAdapter(Type adapterType);
    public static void RegisterDefaultAdapterFactory(
        DataAnnotationsModelValidationFactory factory);

    public static void RegisterDefaultValidatableObjectAdapter(Type adapterType);
    public static void RegisterDefaultValidatableObjectAdapterFactory(
        DataAnnotationsValidatableObjectAdapterFactory factory);
    public static void RegisterValidatableObjectAdapter(Type modelType,
        Type adapterType);
    public static void RegisterValidatableObjectAdapterFactory(Type modelType,
        DataAnnotationsValidatableObjectAdapterFactory factory);
}

```

对于上面的 8 个静态方法，除了 **RegisterDefaultAdapter** 和 **RegisterValidatableObjectAdapter** 之外，其余的都很好理解。**RegisterDefaultAdapter** 用于注册一个默认的针对验证特性的 **ModelValidator** 类型，该类型必须具有一个参数类型列表为 **ModelMetadata**、**ControllerContext** 和 **Attribute** 的构造函数。如果根据验证特性的类型找到了匹配的 **DataAnnotationsModelValidationFactory** 委托对象，相应的参数会被传入该构造函数创建一个我们注册的 **ModelValidator** 对象。

RegisterValidatableObjectAdapter 和 **RegisterDefaultAdapter** 比较类似，用于注册一个默认的针对可验证对象类型的 **ModelValidator**，被注册的 **ModelValidator** 类型必须具有一个参数类型列表为 **ModelMetadata** 和 **ControllerContext** 的构造函数。如果根据验证特性的类型找到了匹配的 **DataAnnotationsValidatableObjectAdapterFactory** 委托对象，相应的参数会被传入该构造函数并最终创建一个我们注册的 **ModelValidator** 对象。

6.3.4 将 ValidationAttribute 应用到参数上

如果你够细心应该会发现我们常用的验证特性都可以直接应用到方法的参数上，下面的 RangeAttribute 的定义为例，应用在该类型上的 AttributeUsageAttribute 的定义表明可以标注该特性的目标元素包括参数、字段和属性。

```
[AttributeUsage(
    AttributeTargets.Parameter |
    AttributeTargets.Field |
    AttributeTargets.Property,
    AllowMultiple=false)]
public class RangeAttribute : ValidationAttribute
{
    //省略成员
}
```

但是对于 ASP.NET MVC 的 Model 验证来说，应用在 Action 方法参数上的验证特性起不到任何作用，因为用于进行 Model 验证的 ModelValidator 对象是通过基于参数类型的 ModelMetadata 来创建的，它根本不会去解析应用在参数本身上的验证特性。

但是在笔者看来，直接针对 Action 方法参数的 Model 验证具有很高的实用价值。一方面，目前的 Model 验证仅限于针对容器对象本身及其属性的验证，如果目标 Action 方法参数为 String、Int32 和 Double 等这样的简单类型，针对它们的验证只能通过手工的方式来完成；另一方面，具有相同参数类型的多个 Action 方法往往具有不同的验证规则。如果我们能够将验证特性应用在参数上进行针对性的验证规则定义，这两个问题都将迎刃而解。

到目前为止，我们对 ASP.NET MVC 的 Model 验证系统已经有了一个全面的了解，现在通过对它进行相应的扩展使直接应用到参数上的验证特性能够生效。我们需要自定义一个 ModelValidatorProvider 来解析应用到参数上的验证特性，并据此生成对应的 ModelValidator。但在这之前需要解决的另一个问题是如何将应用于参数的特性提供给我们自定义的 ModelValidatorProvider，在这里我们将当前 ControllerContext 作为它们的载体。

Action 方法的执行是通过 ActionInvoker 来实现的，默认的 ControllerActionInvoker 和 AsyncControllerActionInvoker 都定义了一个受保护的虚方法 GetParameterValue，它根据描述参数的 ParameterDescriptor 对象和当前的 ControllerContext 借助于 Model 绑定机制得到对应的参数值。我们可以通过继承 ControllerActionInvoker/AsyncControllerActionInvoker 以重写该方法的方式将 ParameterDescriptor 保存当前的 ControllerContext 中。

为此我们自定义了如下两个 ActionInvoker，其中 ParameterValidationActionInvoker 继承自 ControllerActionInvoker，而 ParameterValidationAsyncActionInvoker 是 AsyncControllerActionInvoker 的子类。在重写的 GetParameterValue 方法中，我们在调用基类的同名方法之前将作为参数的 ParameterDescriptor 对象保存到目前 ControllerContext 中，具体来说放到了表示当前路由数据的 RouteDataDictionary 对象的 DataTokens 集合中。在方法调用之后我们将它从 ControllerContext 中移除。

```

public class ParameterValidationActionInvoker : ControllerActionInvoker
{
    protected override object GetParameterValue(
        ControllerContext controllerContext,
        ParameterDescriptor parameterDescriptor)
    {
        try
        {
            controllerContext.RouteData.DataTokens.Add("ParameterDescriptor",
                parameterDescriptor);
            return base.GetParameterValue(controllerContext,
                parameterDescriptor);
        }
        finally
        {
            controllerContext.RouteData.DataTokens.Remove("ParameterDescriptor");
        }
    }
}

public class ParameterValidationAsyncActionInvoker :
    AsyncControllerActionInvoker
{
    protected override object GetParameterValue(ControllerContext
        controllerContext, ParameterDescriptor parameterDescriptor)
    {
        try
        {
            controllerContext.RouteData.DataTokens.Add("ParameterDescriptor",
                parameterDescriptor);
            return base.GetParameterValue(controllerContext,
                parameterDescriptor);
        }
        finally
        {
            controllerContext.RouteData.DataTokens.Remove("ParameterDescriptor");
        }
    }
}

```

被 `ParameterValidationActionInvoker` 和 `ParameterValidationAsyncActionInvoker` 存放到当前 `ControllerContext` 中的 `ParameterDescriptor` 被自定义的 `ModelValidatorProvider` 提取出来用于创建相应的 `ModelValidator`。如下面的代码所示，我们自定义的 `ParameterValidationModelValidatorProvider` 直接继承自 `DataAnnotationsModelValidatorProvider`，在重写的 `GetValidators` 方法中将 `ParameterDescriptor` 从 `ControllerContext` 中提取出来，然后得到应用在参数上的所有特性并与当前的特性列表进行合并，最后将合并的特性列表作为参数调用基类的 `GetValidators` 方法。

```

public class ParameterValidationModelValidatorProvider :
    DataAnnotationsModelValidatorProvider
{
    protected override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context,
        IEnumerable<Attribute> attributes)
    {
        object descriptor;
        if (metadata.ContainerType == null && context.RouteData.DataTokens

```

```

        .TryGetValue("ParameterDescriptor", out descriptor))
    {
        ParameterDescriptor parameterDescriptor =
            (ParameterDescriptor)descriptor;
        DisplayAttribute displayAttribute = parameterDescriptor
            .GetCustomAttributes(true).OfType<DisplayAttribute>()
            .FirstOrDefault()
            ?? new DisplayAttribute
            { Name = parameterDescriptor.ParameterName };
        metadata.DisplayName = displayAttribute.Name;
        var addedAttributes = parameterDescriptor.GetCustomAttributes(true)
            .OfType<Attribute>();
        return base.GetValidators(metadata, context,
            attributes.Union(addedAttributes));
    }
    else
    {
        return base.GetValidators(metadata, context, attributes);
    }
}
}

```

值得一提的是，应用在参数上的特性是针对最外层的容器类型，而不是针对容器类型的属性。比如在类型为 `Contact` 的参数上应用一个验证特性，该特性应该与应用在 `Contact` 类型上的特性具有相同的效果，但是与 `Address` 属性无关。所以 `ParameterDescriptor` 的提取以及特性的合并仅仅在当前 `ModelMetadata` 的 `ContainerType` 为 `Null` 的情况下才会进行。除此之外，我们还利用了标注在参数的 `DisplayAttribute` 特性对 `ModelMetadata` 的 `DisplayName` 属性进行了相应的设置。

在默认的情况下，只有在针对复杂类型的 `Model` 绑定过程中才会进行 `Model` 验证。虽然我们通过 `ParameterValidationModelValidatorProvider` 能够根据应用在 `Action` 方法参数上的验证特性生成相应的 `ModelValidator`，但是如果验证特性是应用在一个简单类型的参数上，对应的 `ModelValidator` 也是不会真正用于参数验证的。为了使 `Model` 验证发生在针对简单类型的 `Model` 绑定过程中，我们不得不创建一个自定义的 `ModelBinder`。

我们定义了一个具有如下定义的 `ParameterValidationModelBinder`，它直接继承自默认使用的 `DefaultModelBinder`，针对简单类型的 `Model` 验证定义在重写的 `BindModel` 方法中。

```

public class ParameterValidationModelBinder : DefaultModelBinder
{
    public override object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        object model = bindingContext.ModelMetadata.Model =
            base.BindModel(controllerContext, bindingContext);
        ModelMetadata metadata = bindingContext.ModelMetadata;
        if (metadata.IsComplexType || null == model)
        {
            return model;
        }

        //针对简单类型的 Model 验证
    }
}

```

```

Dictionary<string, bool> dictionary =
    new Dictionary<string, bool>(StringComparer.OrdinalIgnoreCase);
foreach (ModelValidationResult result in ModelValidator
    .GetModelValidator(metadata, controllerContext).Validate(null))
{
    string key = bindingContext.ModelName;
    if (!dictionary.ContainsKey(key))
    {
        dictionary[key] = bindingContext.ModelState.IsValidField(key);
    }
    if (dictionary[key])
    {
        bindingContext.ModelState.AddModelError(key, result.Message);
    }
}
return model;
}
}

```

到此为止，为了能够将验证特性应用于 Action 方法的参数，我们创建了自定义的 **ActionInvoker**、**ModelValidatorProvider** 和 **ModelBinder**。为了验证它们是否能够最终实现期望的验证效果，我们将它们应用到一个简单的 ASP.NET MVC 应用中。在一个 ASP.NET MVC 应用中创建了一个具有如下定义的 **HomeController**，在重写的 **CreateActionInvoker** 方法中，如果调用基类同名方法返回的 **ActionInvoker** 类型为 **ControllerActionInvoker**，那么将返回一个 **ParameterValidationActionInvoker** 对象，否则返回一个 **ParameterValidationAsyncActionInvoker** 对象，这样做的目的是与默认的同步/异步 Action 执行方式保持一致。

```

public class HomeController : Controller
{
    protected override IActionInvoker CreateActionInvoker()
    {
        IActionInvoker actionInvoker = base.CreateActionInvoker();
        if (actionInvoker is ControllerActionInvoker)
        {
            return new ParameterValidationActionInvoker();
        }
        else
        {
            return new ParameterValidationAsyncActionInvoker();
        }
    }

    public ActionResult Add(
        [Display(Name = "第 1 个操作数")]
        [Range(10, 20, ErrorMessage = "{0}必须在{1}和{2}之间!")]
        [ModelBinder(typeof(ParameterValidationModelBinder))]
        double operand1,

        [Display(Name = "第 2 个操作数")]
        [Range(10, 20, ErrorMessage = "{0}必须在{1}和{2}之间!")]
        [ModelBinder(typeof(ParameterValidationModelBinder))]
        double operand2)
    {
        double result = 0.00;
        if (ModelState.IsValid)

```

```

        {
            result = operand1 + operand2;
        }
        return View(new OperationData { Operand1 = operand1, Operand2 = operand2,
            Operator = "Add", Result = result });
    }
}

public class OperationData
{
    [DisplayName("操作数 1")]
    public double Operand1 { get; set; }
    [DisplayName("操作数 2")]
    public double Operand2 { get; set; }
    [DisplayName("操作符")]
    public string Operator { get; set; }
    [DisplayName("运算结果")]
    public double Result { get; set; }
}

```

我们在 HomeController 中定义了一个进行加法运算的 Action 方法 Add，传入的参数代表两个操作数。我们在两个参数上应用了三个特性，其中 DisplayAttribute 特性用于设置显示名称，验证特性 RangeAttribute 用于限制数值的范围（10 到 20 之间），而 ModelBinderAttribute 则是为了让针对这两个参数的绑定采用我们自定义的 ParameterValidationModelBinder 来完成。在 Add 方法中我们进行相应的运算，将相关的信息封装到一个 OperationData 对象中并作为 Model 呈现在默认的 View 中。

如下所示的是 Action 方法 Add 对应 View 的定义，这是一个 Model 类型为 OperationData 的强类型 View。在该 View 中我们直接调用 HtmlHelper<TModel> 的扩展方法 EditorForModel 将作为 Model 的 OperationData 对象以编辑模式呈现出来。

```

@model OperationData
<html>
    <head>
        <title>ValidationMessage</title>
        <style type="text/css">
            .field-validation-error
            {
                color: Red;
            }
        </style>
    </head>
    <body>
        @Html.EditorForModel()
    </body>
</html>

```

为了让访问 Action 方法 Add 的请求能够直接将操作数放到请求的 URL 中，我们在默认生成的 RouteConfig 类型中作了如下所示的路由注册。然后在 Global.asax 中通过下面的代码对我们自定义的 ParameterValidationModelValidatorProvider 进行了注册，在进行注册之前需要将现有的 ModelValidatorProvider 移除。

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        //其他操作
        routes.MapRoute(
            name: "Add",
            url: "{action}/{operand1}/{operand2}",
            defaults: new { controller = "Home" }
        );
    }
}

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        DataAnnotationsModelValidatorProvider validatorProvider =
            ModelValidatorProviders.Providers
                .OfType<DataAnnotationsModelValidatorProvider>().FirstOrDefault();
        if (null != validatorProvider)
        {
            ModelValidatorProviders.Providers.Remove(validatorProvider);
        }
        ModelValidatorProviders.Providers.Add(
            new ParameterValidationModelValidatorProvider());
    }
}

```

现在运行我们的程序并在浏览器中指定相应的 URL 访问定义在 HomeController 的 Action 方法 Add 进行加法运算，如果提供不合法的操作数（比如 “/Add/50/50”），验证消息将会以如图 6-14 所示的效果显示在相应的文本框旁边。（S613）

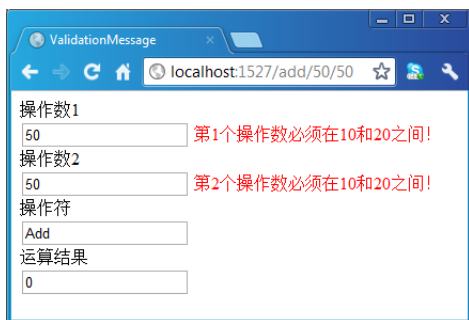


图 6-14 通过应用在 Action 方法参数的验证特性进行 Model 验证

6.3.5 一种 Model 类型，多种验证规则

理想的 Model 验证的应该是场景驱动，而不是类型驱动的，因为同一个数据类型在不同的使用场景中可能具有不同的验证规则。举个简单的例子，对于一个表示应聘者的数据对象

来说，应聘的岗位不同，肯定对应聘者的年龄、性别、专业技能等方面具有不同的要求。

但是 ASP.NET MVC 的 Model 验证恰恰是类型驱动的，因为验证规则是通过应用在数据类型及其属性上的验证特性来定义的，这样的验证方式实际上限制了数据类型在基于不同验证规则的场景中的重用。通过上面的扩展我们将验证特性直接应用在参数上变成了可能，这在一定程度上解决了这个问题，但是也只能解决部分问题，因为应用到参数的验证特性只能用于针对参数类型级别的验证，而不能用于针对参数类型属性级别的验证。

现在我们通过利用对 ASP.NET MVC 的扩展来实现一种基于不同验证规则的 Model 验证。为了让读者对这种认证方式有一个直观的认识，我们先通过一个简单的实例来看看这个扩展最终实现了怎样的验证效果。我们在一个 ASP.NET MVC 应用中定义了如下一个 Person 类型。

```
public class Person
{
    [DisplayName("姓名")]
    public string Name { get; set; }

    [DisplayName("性别")]
    public string Gender { get; set; }

    [DisplayName("年龄")]
    [RangeValidator(10, 20, RuleName="Rule1",
        ErrorMessage = "{0}必须在{1}和{2}之间!")]
    [RangeValidator(20, 30, RuleName = "Rule2",
        ErrorMessage = "{0}必须在{1}和{2}之间!")]
    [RangeValidator(30, 40, RuleName = "Rule3",
        ErrorMessage = "{0}必须在{1}和{2}之间!")]
    public int Age { get; set; }
}
```

在表示年龄的 Age 属性上应用了三个自定义的 RangeValidatorAttribute（不是 RangeAttribute）特性，它们对应着通过 RuleName 属性指定的三种不同的验证规则。三种验证规则（Rule1、Rule2 和 Rule3）分别要求年龄分别在 10~20、20~30 和 30~40 岁之间。

然后我们定义了具有如下定义的 HomeController，它具有三组 Action 方法（Index、Rule1 和 Rule2）。方法 Rule1、Rule2 和 HomeController 类上应用了一个自定义的 ValidationRuleAttribute 特性用于指定当前采用的验证规则。用于指定验证规则的 ValidationRuleAttribute 特性可以同时应用于 Controller 类型和 Action 方法上，后者具有更高的优先级。针对 HomeController 的定义，Action 方法 Index、Rule1 和 Rule2 分别采用的验证规则为 Rule3、Rule1 和 Rule2。

```
[ValidationRule("Rule3")]
public class HomeController : RuleBasedController
{
    public ActionResult Index()
    {
        return View("person", new Person());
    }
}
```



```

[HttpPost]
public ActionResult Index(Person person)
{
    return View("person", person);
}

[ValidationRule("Rule1")]
public ActionResult Rule1()
{
    return View("person", new Person());
}
[HttpPost]
[ValidationRule("Rule1")]
public ActionResult Rule1(Person person)
{
    return View("person", person);
}

[ValidationRule("Rule2")]
public ActionResult Rule2()
{
    return View("person", new Person());
}
[HttpPost]
[ValidationRule("Rule2")]
public ActionResult Rule2(Person person)
{
    return View("person", person);
}
}

```

定义在 HomeController 中的 6 个方法具有相同的操作，它们将创建的/接收的 Person 对象呈现到具有如下定义的 View 中，这是一个 Model 类型为 Person 的强类型 View。在该 View 中我们将作为 Model 的 Person 对象以编辑模式呈现在一个表单中，并在表单中提供一个提交按钮。

```

@model Person
<html>
<head>
    <title>编辑个人信息</title>
    <style type="text/css">
        .field-validation-error{color: red;}
    </style>
</head>
<body>
    @using (Html.BeginForm())
    {
        @Html.EditorForModel()
        <input type="submit" value="保存" />
    }
</body>
</html>

```

现在运行我们的程序，并通过在浏览器中指定相应的地址分别访问定义在 HomeController 的三个 Action (Index、Rule1 和 Rule2)，一个用于编辑个人信息的表单会呈现出来。然后我们根据三个 Action 方法采用的验证规则输入不合法的年龄，然后点击“保存”

按钮，我们会看到输入的年龄按照对应的规则被验证，具体的验证效果如图 6-15 所示。（S614）

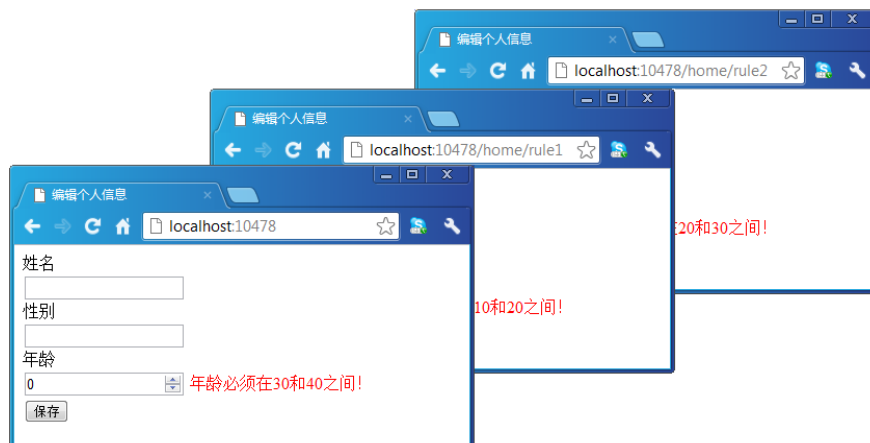


图 6-15 针对不同规则的 Model 验证

我们现在就来具体谈谈上面这个例子所展示的基于不同规则的 Model 验证是如何实现的。首先需要重建一套新的验证特性体系，因为我们需要指定具体的验证规则。定义了一个具有如下定义的抽象 `ValidatorAttribute` 类型，它直接继承自 `ValidationAttribute`，其属性 `RuleName` 表示采用的验证规则名称。我们重写了 `TypeId` 属性，因为需要在相同的属性或者类型上应用多个同类的 `ValidatorAttribute` 特性。

```
[AttributeUsage( AttributeTargets.Class| AttributeTargets.Property,
    AllowMultiple = true)]
public abstract class ValidatorAttribute: ValidationAttribute
{
    private object typeId;
    public string RuleName { get; set; }
    public override object TypeId
    {
        get{return typeId ?? (typeId = new object());}
    }
}
```

上面演示实例中采用的 `RangeValidatorAttribute` 特性定义如下，可以看到它仅仅是对 `RangeAttribute` 的封装。`RangeValidatorAttribute` 具有与 `RangeAttribute` 一致的构造函数定义，并直接使用被封装的 `RangeAttribute` 实施验证。除了能够通过 `RuleName` 指定具体采用的验证规则之外，其他的使用方式与 `RangeAttribute` 完全一致。

```
[AttributeUsage( AttributeTargets.Property, AllowMultiple = true)]
public class RangeValidatorAttribute:ValidatorAttribute
{
    private RangeAttribute rangeAttribute;
    public RangeValidatorAttribute(int minimum, int maximum)
    {
        rangeAttribute = new RangeAttribute(minimum, maximum);
    }
}
```

```

public RangeValidatorAttribute(double minimum, double maximum)
{
    rangeAttribute = new RangeAttribute(minimum, maximum);
}
public RangeValidatorAttribute(Type type, string minimum, string maximum)
{
    rangeAttribute = new RangeAttribute(type, minimum, maximum);
}
public override bool IsValid(object value)
{
    return rangeAttribute.IsValid(value);
}

public override string FormatErrorMessage(string name)
{
    return string.Format(CultureInfo.CurrentCulture, base.ErrorMessageString,
        new object[] { name, rangeAttribute.Minimum,
            rangeAttribute.Maximum });
}
}

```

ValidatorAttribute 的 **RuleName** 属性仅仅指定了验证特性采用的验证规则名称，当前应该采用的验证规则通过应用在 **Action** 方法或者 **Controller** 类型上的 **ValidationRuleAttribute** 特性指定。如下所示的就是这个 **ValidationRuleAttribute** 的定义，验证规则名称通过 **RuleName** 属性表示。

```

[AttributeUsage( AttributeTargets.Class| AttributeTargets.Method)]
public class ValidationRuleAttribute: Attribute
{
    public string RuleName { get; private set; }
    public ValidationRuleAttribute(string ruleName)
    {
        this.RuleName = ruleName;
    }
}

```

对于这个用于实现针对不同验证规则的扩展来说，其核心是如何将通过 **ValidationRuleAttribute** 特性设置的验证规则应用到 **ModelValidator** 的提供机制中，使之筛选出与当前验证规则匹配的验证特性。在这里我们依然使用 **ControllerContext** 来保存这个验证规则名称。细心的读者应该留意到了上面演示实例中创建的 **HomeController** 不是继承自 **Controller**，而是继承自 **RuleBasedController**，这个自定义的 **Controller** 基类定义如下：

```

public class RuleBasedController: Controller
{
    private static Dictionary<Type, ControllerDescriptor>
        controllerDescriptors = new Dictionary<Type, ControllerDescriptor>();
    public ControllerDescriptor ControllerDescriptor
    {
        get
        {
            ControllerDescriptor controllerDescriptor;
            if (controllerDescriptors.TryGetValue(this.GetType(),
                out controllerDescriptor))
            {
                return controllerDescriptor;
            }
        }
    }
}

```

```

    }
    lock (controllerDescriptors)
    {
        if (!controllerDescriptors.TryGetValue(this.GetType(),
            out controllerDescriptor))
        {
            controllerDescriptor =
                new ReflectedControllerDescriptor(this.GetType());
            controllerDescriptors.Add(this.GetType(),
                controllerDescriptor);
        }
        return controllerDescriptor;
    }
}

protected override IAsyncResult BeginExecuteCore(AsyncCallback callback,
    object state)
{
    SetValidationRule();
    return base.BeginExecuteCore(callback, state);
}

protected override void ExecuteCore()
{
    SetValidationRule();
    base.ExecuteCore();
}

private void SetValidationRule()
{
    string actionName = this.ControllerContext.RouteData
        .GetRequiredString("action");
    ActionDescriptor actionDescriptor = this.ControllerDescriptor
        .FindAction(this.ControllerContext, actionName);
    if (null != actionDescriptor)
    {
        ValidationRuleAttribute validationRuleAttribute =
            actionDescriptor.GetCustomAttributes(true)
                .OfType<ValidationRuleAttribute>().FirstOrDefault() ??
            this.ControllerDescriptor.GetCustomAttributes(true)
                .OfType<ValidationRuleAttribute>().FirstOrDefault() ??
            new ValidationRuleAttribute(string.Empty);
        this.ControllerContext.RouteData.DataTokens.Add("ValidationRuleName",
            validationRuleAttribute.RuleName);
    }
}
}

```

在继承自 `Controller` 的 `RuleBasedController` 中, `ExecuteCore` 和 `BeginExecuteCore` 方法被重写。在调用基类的同名方法之前, 我们调用 `SetValidationRule` 方法提取应用在当前 `Action` 方法/`Controller` 类型上的 `ValidationRuleAttribute` 特性指定的验证规则名称, 并将其保存到当前 `ControllerContext` 中。由于针对 `ValidationRuleAttribute` 特性的解析需要使用到用于描述 `Controller` 的 `ControllerDescriptor` 对象, 出于性能考虑, 我们对该对象进行了全局缓存。

对于应用在同一个属性或者类型上的多个基于不同验证规则的 `ValidatorAttribute` 特性, 对应的验证规则名称并没有应用到具体的验证逻辑中。以上面定义的 `RangeValidatorAttribute` 为例, 具体的验证逻辑通过被封装的 `RangeAttribute` 来实现, 如果不做任何处理, 基于不同规则的 `RangeValidatorAttribute` 都将参与到最终的 `Model` 验证过程中。我们必须要做的是在

根据验证特性创建 `ModelValidator` 的时候只选择那些与当前验证规则一致的 `ValidatorAttribute`，这样的操作实现在具有如下定义的 `RuleBasedValidatorProvider` 中。

```
public class RuleBasedValidatorProvider :
    DataAnnotationsModelValidatorProvider
{
    protected override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata, ControllerContext context,
        IEnumerable<Attribute> attributes)
    {
        object validationRuleName = string.Empty;
        context.RouteData.DataTokens.TryGetValue("ValidationRuleName",
            out validationRuleName);
        string ruleName = validationRuleName.ToString();
        attributes = this.FilterAttributes(attributes, ruleName);
        return base.GetValidators(metadata, context, attributes);
    }

    private IEnumerable<Attribute> FilterAttributes(
        IEnumerable<Attribute> attributes, string validationRule)
    {
        var validatorAttributes = attributes.OfType<ValidatorAttribute>();
        var nonValidatorAttributes = attributes.Except(validatorAttributes);
        List<ValidatorAttribute> validValidatorAttributes =
            new List<ValidatorAttribute>();

        if (string.IsNullOrEmpty(validationRule))
        {
            validValidatorAttributes.AddRange(validatorAttributes.Where(
                v => string.IsNullOrEmpty(v.RuleName)));
        }
        else
        {
            var groups = from validator in validatorAttributes
                          group validator by validator.GetType();
            foreach (var group in groups)
            {
                ValidatorAttribute validatorAttribute = group.Where(
                    v => string.Compare(v.RuleName, validationRule, true) ==
                    0).FirstOrDefault();
                if (null != validatorAttribute)
                {
                    validValidatorAttributes.Add(validatorAttribute);
                }
                else
                {
                    validatorAttribute = group.Where(
                        v => string.IsNullOrEmpty(v.RuleName)).FirstOrDefault();
                    if (null != validatorAttribute)
                    {
                        validValidatorAttributes.Add(validatorAttribute);
                    }
                }
            }
        }
        return nonValidatorAttributes.Union(validValidatorAttributes);
    }
}
```

如上面的代码所示, `RuleBasedValidatorProvider` 继承自 `DataAnnotationsModelValidatorProvider`, 基于当前验证规则 (从当前的 `ControllerContext` 中提取) 对 `ValidatorAttribute` 的筛选以及最终对 `ModelValidator` 的创建通过重写的 `GetValidators` 方法实现。具体的筛选机制是: 如果当前的验证规则存在, 则选择与之具有相同规则名称的第一个 `ValidatorAttribute`; 如果这样的 `ValidatorAttribute` 找不到, 则选择第一个没有指定验证规则的 `ValidatorAttribute`; 如果当前的验证规则没有指定, 那么也选择第一个没有指定验证规则的 `ValidatorAttribute`。

我们需要在 `Global.asax` 中通过如下的方式对自定义的 `RuleBasedValidatorProvider` 进行注册, 然后我们的应用就能按照我们期望的方式根据指定的验证规则实施 `Model` 验证了。

```
public class MvcApplication : System.Web.HttpApplication
{
    //其他成员
    protected void Application_Start()
    {
        //其他操作
        DataAnnotationsModelValidatorProvider validator =
            ModelValidatorProviders.Providers
                .OfType<DataAnnotationsModelValidatorProvider>().FirstOrDefault();
        if (null != validator)
        {
            ModelValidatorProviders.Providers.Remove(validator);
        }
        ModelValidatorProviders.Providers.Add(new RuleBasedValidatorProvider());
    }
}
```

6.4 客户端验证

之前我们一直讨论的 `Model` 验证仅限于服务端验证, 即在 `Web` 服务器端根据相应的规则对请求数据实施验证。如果我们能够在客户端 (浏览器) 对用户输入的数据先进行验证, 这样会减少针对服务器请求的频率, 从而缓解 `Web` 服务器访问压力。`ASP.MVC 2.0` 及其之前的版本采用 `ASP.NET Ajax` 进行客户端验证, 在 `ASP.NET MVC 3.0` 中引入了 `jQuery` 验证框架。

6.4.1 jQuery 验证

`Unobtrusive JavaScript` 已经成为了 `JavaScript` 编程的一个指导方针, 但是到目前为止貌似还没有一个针对它的确切定义, 但是一些 `Unobtrusive JavaScript` 的基本原则却已经被广泛地接受。`Unobtrusive JavaScript` 体现了一种被称为“渐进式增强 (PE, `Progressive Enhancement`)”的 `Web` 设计模式, 它采用分层的方式实现了 `Web` 页面内容与功能的分离。用于实现某种功能的 `JavaScript` 不再内嵌于用于展现内容的 `HTML` 中, 而是作为独立的层次建立在 `HTML` 之上。

我们就以验证为例, 假设一个 `Web` 页面中具有如下一个表单, 需要对针对表单中三个

文本框（foo、bar 和 baz）的输入进行验证。假设具体的验证操作实现在 `validate` 函数中，那么我们可以采用如下的 HTML 使相应的文本框在失去焦点的时候对输入的数据实施验证。

```
<form action="/">
  <input id="foo" type="text" onblur="validate()" />
  <input id="bar" type="text" onblur="validate()" />
  <input id="baz" type="text" onblur="validate()" />
  ...
</form>
```

但这不是一个好的设计，理想的方式是让 HTML 只用于定义内容呈现的结构，让 CSS 控制内容呈现的样式，而所有功能的实现定义在独立的 JavaScript 中，所以用于实现验证对 JavaScript 的调用不应该以内联的方式出现在 HTML 中。按照 Unobtrusive JavaScript 的编程方式，我们应该将以内联方式实现的事件注册（`onblur="validate()"`）替换成如下的形式。

```
<form action="/">
  <input id="foo" type="text"/>
  <input id="bar" type="text"/>
  <input id="baz" type="text" />
</form>

<script type="text/javascript">
  window.onload = function () {
    document.getElementById("foo").onblur = validate;
    document.getElementById("bar").onblur = validate;
    document.getElementById("baz").onblur = validate;
  }
</script>
```

Unobtrusive JavaScript 是一个很宽泛的话题，我们不可能在本书中对它进行系统的介绍。Unobtrusive JavaScript 在 jQuery 的验证中得到了很好的体现，接下来我们就简单地介绍一下基于 jQuery 验证的编程。

以内联的方式指定验证规则

jQuery 的验证实际上是对表单中的输入元素进行验证，它支持一种内联的编程方式使我们可以直接将验证的规则定义在被验证表单元素的 `class`（表示 CSS 类型）属性中。考虑到有一些读者对 jQuery 的验证框架可能不太熟悉，为此我们来做一个简单的实例演示。在一个 ASP.NET MVC 应用中定义了如下一个 `HomeController`，在 Action 方法 `Index` 中将默认的 View 呈现出来。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

我们将作为 Web 页面的整个 HTML 定义在 Action 方法 `Index` 对应的 View 中，如下所

示的代码片段是该 View 的定义。

```
<html>
  <head>
    <script type="text/javascript" src="../../Scripts/jquery-1.7.1.js">
    </script>
    <script type="text/javascript" src="../../Scripts/jquery.validate.js">
    </script>
    <script type="text/javascript">
      $(document).ready(function () {
        $("form").validate();
      });
    </script>
    <title>编辑个人信息</title>
  </head>
  <body>
    <form action="/">
      <table>
        <tr>
          <td>姓名: </td>
          <td>
            <input class="required" id="name" name="name" type="text"/>
          </td>
        </tr>
        <tr>
          <td>出生日期: </td>
          <td>
            <input class="required date" id="birhthDate"
              name="birhthDate" type="text"/>
          </td>
        </tr>
        <tr>
          <td>Blog 地址: </td>
          <td>
            <input class="required url" id="blogAddress"
              name="blogAddress" type="text"/>
          </td>
        </tr>
        <tr>
          <td>Email 地址: </td>
          <td>
            <input class="required email" id="emailAddress"
              name="emailAddress" type="text"/>
          </td>
        </tr>
        <tr>
          <td colspan="2">
            <input type="submit" value="保存"/>
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

我们需要将两个必要的.js 文件包含进来, 一个是 jQuery 的核心文件 jquery-1.7.1.js, 另一个是实现验证的 jquery.validate.js。整个 HTML 文件的主体部分是一个表单, 可以通过其

中的文本框输入一些个人信息（姓名、出生日期、Blog 地址和 Email 地址），最后点击“保存”按钮对输入的数据进行提交。

对于这四个文本框对应的<input>元素来说，其 class 属性在这里被用于进行验证规则的定义。其中 required 表示对应的数据是必需的，而 date、url 和 email 则对输入数据的格式进行验证以确保是一个合法的日期、URL 和 Email 地址。真正对输入实施验证体现在如下一段 JavaScript 调用中，在这里我们仅仅是调用<form>元素的 validate 方法而已。

```
<script type="text/javascript">
    $(document).ready(function () {
        $("form").validate();
    });
</script>
```

现在运行我们的程序，一个用于提交个人信息的页面会被呈现出来。当我们输入不合法的数据时（第一次验证发生在提交表单时，之后的验证会在被验证表单元素失去焦点时触发），对应的验证将会自动被触发，而预定义的错误消息将会显示在被验证表单元素的右侧。具体的显示效果如图 6-16 所示。（S615）

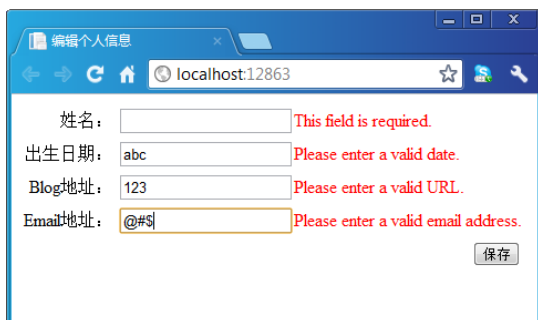


图 6-16 jQuery 验证及其默认错误消息的呈现

单独指定验证规则和错误消息

验证规则其实可以不用以内联的方式定义在被验证表单元素对应的 HTML 中，可以直接将它们定义在用于实施验证的 validate 方法中，该方法不仅仅可以指定表单被验证的输入元素对应的验证规则，还可以指定验证消息，以及控制其他验证行为。

现在我们将上面演示实例中的 View 的 HTML 进行相应的修改，将包含在表单中的四个文本框通过 class 属性设置的验证规则移除。然后在调用表单的 validate 方法实施验证的时候按照如下的方式手工地为被验证输入元素指定相应的验证规则和错误消息。验证规则和错误消息与验证元素之间是通过 name 属性（不是 id 属性）进行关联的。

```
<script type="text/javascript">
    $(document).ready(function () {
        $("form").validate({
            rules: {
                name      : { required: true },
```

```

        birthDate    : { required: true, date: true },
        blogAddress  : { required: true, url: true },
        emailAddress : { required: true, email: true }
    },

    messages: {
        name          : { required: "请输入姓名" },
        birthDate     : { required: "请输入出生日期",
                        date: "请输入一个合法的日期" },
        blogAddress   : { required: "请输入 Blog 地址",
                        url: "请输入一个合法的 URL" },
        emailAddress  : { required: "请输入 Email 地址",
                        email: "请输入一个合法的 Email 地址" }
    }
});
});
</script>

```

再次运行我们的程序后发现，定制的错误消息就会按照如图 6-17 所示的效果呈现出来。(S616)



图 6-17 jQuery 验证及其定制错误消息的呈现

6.4.2 基于 jQuery 的 Model 验证

在简单了解了 Unobtrusive JavaScript 形式的验证在 jQuery 中的实现之后，我们来具体讨论 ASP.NET MVC 是如何利用它实现客户端验证的。服务端验证最终实现在相应的 `ModelValidator` 中，而最终的验证规则定义在相应的 `ValidationAttribute` 中，而客户端验证规则通过 `HtmlHelper<TModel>` 相应的模板方法（比如 `TextBoxFor`、`EditorFor` 和 `EditorForModel` 等）输出到生成的 HTML 中。服务端验证和客户端验证必须采用相同的验证规则，通过应用 `ValidationAttribute` 特性定义的验证规则也同样体现在基于客户端验证规则的 HTML 上。

ValidationAttribute 与 HTML

ASP.NET MVC 默认采用基于验证特性的声明式验证，服务端验证最终实现在两个重写

的 `IsValid` 方法中。对于客户端验证，ASP.NET MVC 对 jQuery 的验证插件进行了扩展，它将验证规则以表单元素属性的方式输出到最终生成的 HTML 中。为了让客户端和服务端采用相同的验证规则，应用在 Model 类型某个属性上的 `ValidationAttribute` 特性最终会反映在被验证表单元素对应的 HTML 上。

```
public class Contact
{
    [DisplayName("姓名")]
    [Required(ErrorMessage = "请输入{0}!")]
    [StringLength(8, ErrorMessage = "作为{0}字符串长度不能超过{1}!")]
    public string Name { get; set; }

    [DisplayName("电子邮箱地址")]
    [RegularExpression(@"^\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}$",
        ErrorMessage = "请输入正确的电子邮箱地址!")]
    public string EmailAddress { get; set; }
}
```

假设我们具有如上一个数据类型 `Contact`，`RequiredAttribute` 和 `StringLengthAttribute` 特性应用到表示姓名的 `Name` 属性上用于确保用户必须输入一个不超过 128 个字符的字符串，而表示 Email 地址的 `EmailAddress` 属性应用了一个 `RegularExpressionAttribute` 用于确保用户输入一个合法的 Email 地址。在一个以此 `Contact` 为 Model 类型的 View 中，如果我们调用 `HtmlHelper<TModel>` 的扩展方法 `EditorForModel` 最终会生成如下一段 HTML。

```
<div class="editor-label">
    <label for="Name">姓名</label>
</div>

<div class="editor-field">
    <input class="text-box single-line"
        data-val          ="true"
        data-val-length    ="作为姓名字符串长度不能超过 8!"
        data-val-length-max ="8"
        data-val-required  ="请输入姓名!"
        id="Name" name="Name" type="text" value="" />
    <span class="field-validation-valid" data-valmsg-for="Name"
        data-valmsg-replace="true"></span>
</div>

<div class="editor-label">
    <label for="EmailAddress">电子邮箱地址</label>
</div>

<div class="editor-field">
    <input class="text-box single-line"
        data-val          ="true"
        data-val-regex     ="请输入正确的电子邮箱地址! "
        data-val-regex-pattern="^\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}$"
        id="EmailAddress" name="EmailAddress" type="text" value="" />
    <span class="field-validation-valid" data-valmsg-for="EmailAddress"
        data-valmsg-replace="true"></span>
</div>
```

通过上面的这段 HTML 我们可以看到, Contact 的两个属性对应的<input>元素具有一个“data-val”属性和一系列以“data-val-”为前缀的属性,前者表示是否需要对用户输入的值进行验证,后者则代表相应的验证规则。具体来说,去除“data-val-”前缀后的属性名称就是 jQuery 验证规则名称。

一般来说,一个 ValidationAttribute 对应着一种验证类型和一系列可选的验证参数。比如 RequiredAttribute、StringLengthAttribute 和 RegularExpressionAttribute 对应的验证类型分别是“required”、“length”和“regex”,而 StringLengthAttribute 和 RegularExpressionAttribute 各自具有一个验证参数 length-max (表示允许的字符串最大长度)和 regex-pattern (正则表达式)。验证错误消息一般作为验证类型属性的值,而验证参数对应的属性值自然就是相应的参数值。

对于上面生成的 HTML 还有一点值得一提,对应着被验证属性的<input>元素会紧跟一个元素用于显示验证失败后的错误消息。该元素的 CSS 类型为“field-validation-valid”,当验证失败后被替换为“field-validation-error”,可以通过它来定制错误消息的显示样式。

客户端验证规则的生成

ASP.NET MVC 在利用 jQuery 进行客户端验证的时候,虽然验证规则并没有采用其原生的方式通过被验证元素的 class 属性来提供,但是却可以通过“data-val-{rulename}”的命名模式提取相应的验证规则属性值,并最终得到对应的验证规则,ASP.NET MVC 只需要对此作简单的“适配”即可。我们现在关心的是当调用定义在 HtmlHelper<TModel>中相应的模板方法将 Model 对象的某个属性以表单元素的形式进行呈现的时候,ASP.NET MVC 是如何生成这些以“data-val-”为前缀的验证属性的呢?

在这里我们需要涉及到一个表示客户端验证规则的 ModelClientValidationRule 类型。如下面的代码所示,ModelClientValidationRule 具有三个属性,字符串属性 ErrorMessage 和 ValidationType 表示验证错误消息和验证的类型,类型为 IDictionary<string, object>的只读属性 ValidationParameters 表示辅助客户端验证的参数,其中 Key 和 Value 分别表示验证参数名和参数值。

```
public class ModelClientValidationRule
{
    public string ErrorMessage { get; set; }
    public string ValidationType { get; set; }
    public IDictionary<string, object> ValidationParameters { get; }
}

public abstract class ModelValidator
{
    //其他成员
    public virtual IEnumerable<ModelClientValidationRule>
```

```

        GetClientValidationRules();
        public abstract IEnumerable<ModelValidationResult> Validate(
            object container);
    }

```

通过前面的介绍我们知道，抽象类 `ModelValidator` 中具有一个虚方法 `GetClientValidationRules`，它用于创建一个 `ModelClientValidationRule` 对象的列表，所有支持客户端验证的 `ModelValidator` 必须重写该方法以生成相应的客户端验证规则。

以用于进行范围验证的 `RangeAttribute` 特性对应的 `RangeAttributeAdapter` 为例，如下面的代码片段所示，它重写了 `GetClientValidationRules`，返回的 `ModelClientValidationRule` 列表中包含一个 `ModelClientValidationRangeRule` 对象。该 `ModelClientValidationRangeRule` 对象的验证类型为“range”，采用 `RangeAttributeAdapter` 的 `ErrorMessage` 属性作为自身的错误消息。作为验证范围的上、下限的两个属性（`Maximum` 和 `Minimum`）成为了该 `ModelClientValidationRule` 的两个验证参数，参数名分别为“max”和“min”。

```

public class RangeAttributeAdapter :
    DataAnnotationsModelValidator<RangeAttribute>
{
    //其他成员
    public override IEnumerable<ModelClientValidationRule>
        GetClientValidationRules()
    {
        string errorMessage = base.ErrorMessage;
        return new ModelClientValidationRangeRule[] {
            new ModelClientValidationRangeRule(errorMessage,
                base.Attribute.Minimum, base.Attribute.Maximum) };
    }
}

public class ModelClientValidationRangeRule : ModelClientValidationRule
{
    public ModelClientValidationRangeRule(string errorMessage, object minValue,
        object maxValue)
    {
        base.ErrorMessage = errorMessage;
        base.ValidationType = "range";
        base.ValidationParameters["min"] = minValue;
        base.ValidationParameters["max"] = maxValue;
    }
}

```

客户端验证在这里还涉及到一个重要的接口 `System.Web.Mvc.IClientValidatable`，它具有唯一的 `GetClientValidationRules` 方法来返回一个以 `System.Web.Mvc.ModelClientValidationRule` 对象表示的客户端验证规则列表。

```

public interface IClientValidatable
{
    IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context);
}

```

所有支持客户端验证的 `ValidationAttribute` 都需要实现 `IClientValidatable` 接口并通过实现

`GetClientValidationRules` 方法提供对应的验证规则，而生成的验证规则需要与通过重写的 `IsValid` 方法实现的服务端验证逻辑一致。`DataAnnotationsModelValidator` 重写了 `GetClientValidationRules` 方法，如果对应的 `ValidationAttribute` 实现了 `IClientValidatable` 接口，它（`ValidationAttribute`）的 `GetClientValidationRules` 方法被调用并将返回的 `ModelClientValidationRule` 列表作为该方法的返回值。

当我们在某个 View 中调用 `HtmlHelper<TModel>` 的模板方法将 Model 对象的某个属性以表单元素呈现出来的时候，它会采用我们前面介绍的 `ModelValidator` 的提供机制根据目标属性对应的 `ModelMetadata` 创建相应的 `ModelValidator`，然后调用 `GetClientValidationRules` 方法得到一组表示客户端验证规则的 `ModelClientValidationRule` 列表。如果该列表不为空，它们将作为验证属性附加到目标属性对应的 `<input>` 元素中。

6.4.3 自定义验证

虽然在命名空间 `System.ComponentModel.DataAnnotations` 中具有一系列继承自 `ValidationAttribute` 的验证特性可以帮助我们完成基本的数据验证，但是在很多场景下的验证需要按照我们自定义的方式进行，接下来以实例演示的方式来指导读者如何以自定义的方式实现服务端和客户端验证。

假设需要对表示出生日期的输入数据进行验证以确保其年龄在允许的范围内，为此我们创建了一个自定义的验证特性 `AgeRangeAttribute`。如下面的代码片段所示，`AgeRangeAttribute` 应用在表示出生日期的属性上并且指定允许年龄范围（18~25）的上下限。

```
public class Person
{
    [DisplayName("姓名")]
    public string Name { get; set; }

    [DisplayName("出生日期")]
    [AgeRange(18, 25, ErrorMessage = "年龄必须在{0}到{1}周岁之间!")]
    [DisplayFormat(ApplyFormatInEditMode = true,
        DataFormatString = "{0:yyyy-MM-dd}")]
    public DateTime? BirthDate { get; set; }
}
```

为简单起见，我们直接让 `AgeRangeAttribute` 继承自 `RangeAttribute`。如下面的代码片段所示，我们在构造函数中以整数的形式指定表示年龄范围的下限和上限。服务端验证体现在重写的 `IsValid` 方法中，而为了让格式化的错误消息是针对年龄而不是针对出生日期，我们重写了 `FormatErrorMessage` 方法。

```
[AttributeUsage(AttributeTargets.Property)]
public class AgeRangeAttribute: RangeAttribute, IClientValidatable
{
    public AgeRangeAttribute(int minimum, int maximum)
        : base(minimum, maximum)
```

```

{ }

public override bool IsValid(object value)
{
    DateTime? birthDate = value as DateTime?;
    if (null == birthDate)
    {
        return true;
    }
    DateTime age = new DateTime(DateTime.Today.Ticks -
        birthDate.Value.Ticks);
    return (int)this.Minimum <= age.Year && age.Year <= (int)this.Maximum;
}

public override string FormatErrorMessage(string name)
{
    return string.Format(CultureInfo.CurrentCulture, this.ErrorMessageString,
        this.Minimum, this.Maximum);
}

public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
    ModelMetadata metadata, ControllerContext context)
{
    string errorMessage = FormatErrorMessage("");
    ModelClientValidationRule rule = new ModelClientValidationRule
    {
        ValidationType = "agerange", ErrorMessage = errorMessage
    };
    rule.ValidationParameters.Add("minage", this.Minimum);
    rule.ValidationParameters.Add("maxage", this.Maximum);
    yield return rule;
}
}

```

AgeRangeAttribute 实现了 IClientValidatable 接口，在 GetClientValidationRules 方法中返回一个验证类型为“agerange”的 ModelClientValidationRule 对象。它具有两个验证参数（minage 和 maxage）分别表示年龄范围的下限和上限。通过调用 FormatErrorMessage 方法格式化后生成的消息成为了该 ModelClientValidationRule 的错误消息。

ModelClientValidationRule 对象的 ValidationType 属性值最终表示 jQuery 验证插件的验证规则，而该规则通过一个对应的函数来实施验证。对于 AgeRangeAttribute 来说，其对应的客户端验证类型为“agerange”，为此我们在一个.js 文件中以此命名注册的验证函数，具体的定义如下所示。

```

jQuery.validator.addMethod("agerange",function (value, element, params) {
    value = value.replace(/(^\\s*)(\\s*$)/g, "");
    if (!value) {
        return true;
    }
    var minAge = params.minage;
    var maxAge = params.maxage;

    var birthDateArray = value.split("-");
    var birthDate = new Date(birthDateArray[0], birthDateArray[1],
        birthDateArray[2]);
    var currentDate = new Date();
    var age = currentDate.getFullYear() - birthDate.getFullYear();
    return age >= minAge && age <= maxAge;
}

```

```
});

jQuery.validator.unobtrusive.adapters.add("agerange", ["minage", "maxage"],
function (options) {
    options.rules["agerange"] = {
        minage: options.params.minage,
        maxage: options.params.maxage
    };
    options.messages["agerange"] = options.message;
});
```

如上面的代码片段所示，我们调用全局对象 jQuery 的 validator 属性的 AddMethod 方法添加了一个用于进行年龄范围验证的函数，该函数对应的验证规则为“agerange”。验证函数具有三个参数，其中 value 和 element 分别代表被验证的数据值（表示出生日期的字符串）和 HTML 元素，而 params 参数则表示传入的验证参数（表示年龄范围的上、下限）。最后我们调用 jQuery.validator.unobtrusive.adapters 的 add 方法对验证规则 agerange 进行注册，并指定对应的验证参数名称列表。

我们在一个 ASP.NET MVC 应用中定义了以前面定义的 Person 类型为 Model 的 View。如下面的代码片段所示，我们调用 HtmlHelper<TModel>的扩展方法 EditorForModel 将作为 Model 的 Person 对象以编辑模式呈现在一个表单之中，通过<script>标签引用了四个.js 文件，前三个是 ASP.NET MVC 原生脚本，最后一个是我们自定义的脚本文件，上面定义的“agerange”相关的扩展就定义在这个文件中。

```
@model Person
<html>
    <head>
        <script type="text/javascript"
            src="@Url.Content("~/Scripts/jquery-1.7.1.js")"></script>
        <script type="text/javascript"
            src="@Url.Content("~/Scripts/jquery.validate.js")"></script>
        <script type="text/javascript"
            src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.js")">
            </script>
        <script type="text/javascript"
            src="@Url.Content("~/Scripts/jquery.validator.extend.js")">
            </script>
        <style type="text/css">
            .field-validation-error {color:Red}
        </style>
        <title>编辑个人信息</title>
    </head>
    <body>
        @using(Html.BeginForm())
        {
            @Html.EditorForModel()
            <input type="submit" value="保存" />
        }
    </body>
</html>
```

运行程序并在浏览器中将该 View 呈现出来，在输入违反年龄限制的出生日期的时候，客户端验证将会生效并以图 6-18 所示的形式显示出错误消息。

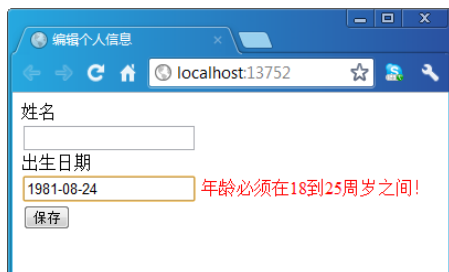


图 6-18 自定义验证对错误消息的呈现

如果我们查看最终生成出来的 HTML，会发现表示出生日期的文本框具有如下所示的定义，高亮显示的三个以“data-val-”为前缀的属性内容正是根据 `AgeRangeAttribute` 特性的 `GetClientValidationRules` 方法返回的 `ModelClientValidationRule` 对象生成的。另一个名为“data-val-date”的属性是默认添加的针对日期格式的验证规则，由于没有引用相应的本地化 JavaScript 文件，错误消息被默认设置为英文。

```
<input class="text-box single-line"
      data-val="true"
      data-val-agerange="年龄必须在 18 到 25 周岁之间! "
      data-val-agerange-maxage="25"
      data-val-agerange-minage="18"
      data-val-date="The field 出生日期 must be a date."
      id="BirthDate" name="BirthDate" type="text" value="" />
```

本章小结

Model 验证旨在为通过 Model 绑定生成的参数进行检验以确保用户输入数据的有效性。`ModelValidator` 是整个 Model 验证系统的核心，ASP.NET MVC 定义了 `DataAnnotationsModelValidator`、`ValidationErrorModelValidator` 和 `ValidatableObjectAdapter` 三种 `ModelValidator` 类型，它们体现了实现 Model 验证的三种解决方案。每一种 `ModelValidator` 均由对应的 `ModelValidatorProvider` 来提供，而自定义的 `ModelValidatorProvider` 通过静态类型 `ModelValidatorProviders` 进行注册。

Model 验证是伴随着 Model 绑定进行的，而且两者最终都涉及到对 `ModelState` 的操作，Model 绑定将 `ValueProvider` 提供的数据写入 `ModelState` 中，而 Model 验证则将验证后的错误消息保存到 `ModelState` 中。正是由于验证的结构最终体现在当前的 `ModelState` 中，可以利用 `HtmlHelper/HtmlHelper<TModel>` 相应的扩展方法将相应的验证错误消息呈现在 View 中。

基于验证特性的声明式 Model 验证通过 `DataAnnotationsModelValidator` 来实现。在默认的情况下，只有应用在自定义容器类型及其属性上的验证特性才会生效，不过我们可以通过扩展将验证特性直接应用到 Action 方法的参数上。同样是利用 Model 验证系统的扩展，我们实现了利用验证特性在同一个数据类型上定义不同的验证规则。

除了服务端验证，ASP.NET MVC 还支持针对 ASP.NET Ajax 和 jQuery 的客户端验证。客户端验证规则通过 `ModelClientValidationRule` 对象表示，ASP.NET MVC 利用调用 `ModelValidator` 的 `GetClientValidationRules` 方法获得的 `ModelClientValidationRule` 列表辅助 `HtmlHelper<TModel>` 的模板方法生成与验证相关的 HTML。需要进行客户端验证的自定义 `ValidationAttribute` 可以通过实现 `IClientValidatable` 接口的方式来提供定义了验证规则的 `ModelClientValidationRule` 列表。

第 7 章 Action 的执行

对于一个 ASP.NET MVC 应用来说，一个请求的目标通常是定义在某个 Controller 中的 Action 方法，所以对请求的处理最终体现在对目标 Action 方法的执行。本章讨论的“Action 的执行”不仅仅包括 Action 方法本身的执行，还包括相关筛选器的执行。

7.1 异步 Action 的定义

ASP.NET MVC 3 中的异步 Action 通过两个匹配的方法 `XxxAsync/XxxCompleted` 来定义，这样的异步 Action 只能定义在继承自 `AsyncController` 的类型中。ASP.NET MVC 4 提供了一种更加简洁的异步 Action 定义方式，不过为了兼容前一版本，传统的定义方式和 `AsyncController` 类型依然被保留下来。为什么我们需要让 Action 方法异步地执行呢？要回答这个问题，这就需要了解 ASP.NET 基于线程池（Thread Pool）的请求处理机制。

7.1.1 基于线程池的请求处理机制

ASP.NET 通过线程池的机制处理并发的 HTTP 请求。一个 Web 应用内部维护着一个线程池，当探测到抵达的请求后，ASP.NET 会从池中获取一个空闲的线程来处理它。处理完后，线程不会被回收，而是重新释放到池中。线程池具有一个最大容量，如果创建的线程达到这个上限并且现有的线程均被处于“忙碌”状态，新的 HTTP 请求会被放入一个请求队列以等待某个线程重新释放到池中。

我们将这些用于处理 HTTP 请求的线程称为工作线程（Worker Thread），而这个线程池自然就叫做工作线程池。ASP.NET 这种基于线程池的请求处理机制主要具有如下两个优势。

- 工作线程的重用：创建线程的成本虽然不如进程的激活，却也不是一件一蹴而就的事情，频繁地创建和释放线程会对性能造成极大的损害。线程池机制避免了总是创建新的工作线程来处理每一个请求，被创建的工作线程得到了极大地重用，并最终提高了服务器的吞吐能力。
- 工作线程数量的限制：资源的有限性决定了服务器处理请求的能力具有一个上限，或者说某台服务器能够处理的请求并发量具有一个临界点，一旦超过这个临界点，整个服务器将会因不能提供足够的资源而崩溃。由于采用了对工作线程数量具有良好控制的线程池机制，ASP.NET 并发处理的请求数量不可能超过线程池的最大允许的容量，从而避免了在高并发情况下工作线程的无限制创建而最终导致整个服务器的崩溃。

如果请求处理操作耗时较短，工作线程处理完后可以及时地被释放到线程池中以用于对下一个请求的处理。但是对于比较耗时的操作来说，意味着工作线程将长时间被某个请求独占，如果这样的操作访问比较频繁，在高并发的情况下在线程池中可能找不到空闲的工作线程用于及时处理最新抵达的请求。

如果采用异步的方式来处理这样的耗时请求，工作线程可以让后台线程来接手，而自己可以及时地被释放到线程池中用于进行后续请求的处理，从而提高了整个服务器的吞吐能力。值得一提的是，异步操作主要用于 I/O 绑定操作（比如数据库访问和远程服务调用等），而非 CPU 绑定操作，因为异步操作对整体性能的提升来源于当 I/O 设备在处理某个任务的

时候，CPU 可以释放出来去处理另一个任务。如果耗时操作主要依赖于本机 CPU 的运算，采用异步方法反而会因为线程调度和线程上下文的切换而影响整体的性能。

7.1.2 两种异步 Action 方法的定义

在了解了异步 Action 的必要性之后，我们来简单介绍一下异步 Action 的定义方式。总的来说，异步 Action 方法具有两种定义方式，一种是将其定义成两个匹配的方法 XxxAsync/XxxCompleted，另一种则是定义一个返回类型为 Task 的方法。

XxxAsync/XxxCompleted

如果使用两个匹配的方法 XxxAsync/XxxCompleted 来定义异步 Action，可以将异步操作实现在 XxxAsync 方法中，而将最终内容的响应实现在 XxxCompleted 方法中。XxxCompleted 可以看成是对 XxxAsync 方法的回调，当定义在 XxxAsync 方法中的操作以异步方式完成执行后，XxxCompleted 方法会被自动调用。XxxCompleted 的定义方式和普通的同步 Action 方法比较类似。

作为演示，笔者在如下所示的 HomeController 中定义了一个名为 Article 的异步操作来显示指定名称的文章内容。我们将内容的异步读取定义在 ArticleAsync 方法中，ArticleCompleted 方法负责将读取的内容以 ContentResult 的形式呈现出来。（S701）

```
public class HomeController : AsyncController
{
    public void ArticleAsync(string name)
    {
        AsyncManager.OutstandingOperations.Increment();
        Task.Factory.StartNew(() =>
        {
            string path = ControllerContext.HttpContext.Server
                .MapPath(string.Format(@"\articles\{0}.html", name));
            using (StreamReader reader = new StreamReader(path))
            {
                AsyncManager.Parameters["content"] = reader.ReadToEnd();
            }
            AsyncManager.OutstandingOperations.Decrement();
        });
    }

    public ActionResult ArticleCompleted(string content)
    {
        return Content(content);
    }
}
```

对于以 XxxAsync/XxxCompleted 形式定义的异步 Action 方法来说，ASP.NET MVC 并不会以异步的方式来调用 XxxAsync 方法，所以我们需要在该方法中自行实现异步。在上面定义的 ArticleAsync 方法中，我们是通过基于 Task 的并行编程方式来实现对文章内容的异步

读取的。

当我们以 `XxxAsync/XxxCompleted` 形式定义异步 Action 方法的时候，会频繁地使用到 Controller 的 `AsyncManager` 属性，该属性返回一个类型为 `System.Web.Mvc.Async.AsyncManager` 对象，将在下面一节对其进行单独讲述。

在上面提供的实例中，我们在异步操作开始和结束的时候调用了 `AsyncManager` 的 `OutstandingOperations` 属性的 `Increment` 和 `Decrement` 方法对 ASP.NET MVC 发起异步操作开始和结束的通知。我们还利用 `AsyncManager` 的 `Parameters` 属性表示的字典来保存传递给 `ArticleCompleted` 方法的参数，参数在字典中的对应的 Key（“content”）可以与 `ArticleCompleted` 的参数名称自动匹配，所以在调用方法 `ArticleCompleted` 的时候，通过 `AsyncManager` 的 `Parameters` 属性指定的对象将自动作为对应的参数值。

Task 返回值

如果采用上面的异步 Action 定义方式，意味着我们不得不为一个 Action 定义两个方法，实际上可以通过一个方法来完成对异步 Action 的定义，那就是让 Action 方法返回一个代表异步操作的 Task 对象。除此之外，以 `XxxAsync/XxxCompleted` 形式定义的异步 Action 只能出现在继承自 `AsyncController` 的类型中，而针对 Task 返回值的异步 Action 则对此没有限制。实际上保留 `AsyncController` 这个抽象类主要是为了实现对 ASP.NET MVC 3 的向后兼容。上面通过 `XxxAsync/XxxCompleted` 形式定义的异步 Action 可以采用如下的定义方式。（S702）

```
public class HomeController : Controller
{
    public Task<ActionResult> Article(string name)
    {
        return Task.Factory.StartNew(() =>
        {
            string path = ControllerContext.HttpContext.Server
                .MapPath(string.Format(@"\articles\{0}.html", name));
            using (StreamReader reader = new StreamReader(path))
            {
                AsyncManager.Parameters["content"] = reader.ReadToEnd();
            }
        }).ContinueWith<ActionResult>(task =>
        {
            string content = (string)AsyncManager.Parameters["content"];
            return Content(content);
        });
    }
}
```

上面定义的异步 Action 方法 `Article` 返回一个 `Task<ActionResult>` 对象，异步文件内容的读取体现在返回的 Task 对象中。对文件内容呈现的回调操作则通过调用该 Task 对象的 `ContinueWith<ActionResult>` 方法进行注册，该操作会在异步操作完成之后被自动调用。

如上面的代码片段所示，我们依然利用 `AsyncManager` 的 `Parameters` 属性实现参数在异步操作和回调操作之间的传递。其实我们也可以使用 Task 对象的 `Result` 属性来实现相同的

功能，Article 方法的定义也改写成如下的形式。

```
public class HomeController : AsyncController
{
    public Task<ActionResult> Article(string name)
    {
        return Task.Factory.StartNew(() =>
        {
            string path = ControllerContext.HttpContext.Server
                .MapPath(string.Format(@"\articles\{0}.html", name));
            using (StreamReader reader = new StreamReader(path))
            {
                return reader.ReadToEnd();
            }
        }).ContinueWith<ActionResult>(task =>
        {
            return Content((string)task.Result);
        });
    }
}
```

7.1.3 AsyncManager

在上面演示的异步 Action 的定义中，我们通过 AsyncManager 实现了两个基本的功能，即在异步操作和回调操作之间传递参数和向 ASP.NET MVC 发送异步操作开始和结束的通知。由于 AsyncManager 在异步 Action 场景中具有重要的作用，我们有必要对其进行单独介绍，如下所示是 AsyncManager 的定义。

```
public class AsyncManager
{
    public AsyncManager();
    public AsyncManager(SynchronizationContext syncContext);

    public EventHandler Finished;

    public virtual void Finish();
    public virtual void Sync(Action action);

    public OperationCounter OutstandingOperations { get; }
    public IDictionary<string, object> Parameters { get; }
    public int Timeout { get; set; }
}

public sealed class OperationCounter
{
    public event EventHandler Completed;

    public int Increment();
    public int Increment(int value);
    public int Decrement();
    public int Decrement(int value);

    public int Count { get; }
}
```

如上面的代码片段所示, `AsyncManager` 具有两个构造函数重载, 非默认构造函数接受一个表示同步上下文的 `SynchronizationContext` 对象作为参数。如果指定的同步上下文对象为 `Null`, 并且当前的同步上下文 (通过 `SynchronizationContext` 的静态属性 `Current` 表示) 存在, 则使用当前上下文, 否则创建一个新的同步上下文。该同步上下文用于 `Sync` 方法的执行, 也就是说在该方法指定的 `Action` 委托将会在该同步上下文中以同步的方式执行。

`AsyncManager` 的核心是通过属性 `OutstandingOperations` 表示的正在进行的异步操作计数器, 该属性是一个类型为 `System.Web.Mvc.Async.OperationCounter` 的对象。操作计数通过只读属性 `Count` 表示, 当我们开始和完成异步操作的时候分别调用 `Increment` 和 `Decrement` 方法作增加和减少计数操作。`Increment` 和 `Decrement` 方法各自具有两个重载, 作为整数参数 `value` (该参数值可以是负数) 表示增加或者减少的数值, 如果调用无参方法, 增加或者减少的数值为 1。如果我们需要同时执行多个异步操作, 则可以通过如下的方法来操作计数器。

```
AsyncManager.OutstandingOperations.Increment(3);

Task.Factory.StartNew(() =>
{
    //异步操作 1
    AsyncManager.OutstandingOperations.Decrement();
});
Task.Factory.StartNew(() =>
{
    //异步操作 2
    AsyncManager.OutstandingOperations.Decrement();
});
Task.Factory.StartNew(() =>
{
    //异步操作 3
    AsyncManager.OutstandingOperations.Decrement();
});
```

对于每次通过 `Increment` 和 `Decrement` 方法调用引起的计数数值的改变, `OperationCounter` 对象都会检验当前计数数值是否为零, 为零则表明所有的操作运行完毕, 在这种情况下 `Completed` 事件会被触发。值得一提的是, 表明所有操作完成执行的标志是计数器的值等于零, 而不是小于零, 如果我们通过调用 `Increment` 和 `Decrement` 方法使计数器的值成为一个负数, 注册的 `Completed` 事件是不会被执行的。

`AsyncManager` 在初始化时就注册了 `OperationCounter` 对象 (对应 `OutstandingOperations` 属性) 的 `Completed` 事件, 使该事件触发的时候调用自身的 `Finish` 方法。实现在 `AsyncManager` 的虚方法 `Finish` 又会触发自身的 `Finished` 事件。也就是说, `Finished` 事件最终会在异步操作计数器变为零时被触发。

如下面的代码片段所示, `Controller` 类实现了 `System.Web.Mvc.Async.IAsyncManagerContainer` 接口, 而后者定义了一个只读属性 `AsyncManager` 用于提供辅助执行异步 `Action` 的 `AsyncManager` 对象。我们在定义异步 `Action` 方法时使用的 `AsyncManager` 对象就是从抽象类 `Controller` 中继承下来的 `AsyncManager` 属性值。

```

public abstract class Controller : ControllerBase, IAsyncManagerContainer, ...
{
    public AsyncManager AsyncManager { get; }
}

public interface IAsyncManagerContainer
{
    AsyncManager AsyncManager { get; }
}

```

XxxCompleted 方法的执行

对于通过 XxxAsync/XxxCompleted 形式定义的异步 Action，回调操作 XxxCompleted 会在定义于 XxxAsync 方法中的异步操作执行结束之后被自动调用，那么 XxxCompleted 方法具体是如何被执行的呢？

通过下一节的介绍我们将会知道异步 Action 的执行最终是通过描述它的 AsyncActionDescriptor 对象的 BeginExecute/EndExecute 方法来完成的。通过第 5 章“Model 的绑定”的介绍我们知道以 XxxAsync/XxxCompleted 形式定义的异步 Action 是通过一个 ReflectedAsyncActionDescriptor 对象来表示的，它在执行 BeginExecute 方法的时候会注册 Controller 对象的 AsyncManager 的 Finished 事件，让该事件触发的时候去执行 Completed 方法。

也就是说 AsyncManager 的 Finished 事件的触发标志着异步操作的结束，而此时匹配的 Completed 方法会被执行。由于 AsyncManager 的 Finish 方法会主动触发该事件，所以我們也可以通过调用该方法促使 Completed 方法立即执行。由于 AsyncManager 的 OperationCounter 对象的 Completed 事件触发的时候会调用 Finish 方法，所以当表示当前正在执行的异步操作计算器的值为零时，Completed 方法也会自动被执行。

如果我们在 XxxAsync 方法中通过如下的方式同时执行三个异步操作，并在每个操作完成之后调用 AsyncManager 的 Finish 方法，意味着最先完成的异步操作会导致 XxxCompleted 方法的执行。换句话说，当 XxxCompleted 方法执行的时候，可能还有两个异步操作正在执行。

```

AsyncManager.OutstandingOperations.Increment(3);

Task.Factory.StartNew(() =>
{
    //异步操作 1
    AsyncManager.Finish();
});
Task.Factory.StartNew(() =>
{
    //异步操作 2
    AsyncManager.Finish();
});
Task.Factory.StartNew(() =>
{
    //异步操作 3
    AsyncManager.Finish();
});

```

如果完全通过异步操作计数机制来控制 `XxxCompleted` 方法的执行，由于计数的检测和 `Completed` 事件的触发只发生在 `OperationCounter` 的 `Increment/Decrement` 方法被执行的时候，倘若我们在开始和结束异步操作的时候都没有调用这两个方法，`XxxCompleted` 是否会执行呢？同样以之前定义的用于读取/显示文章内容的异步 Action 为例，按照如下的方式将定义在 `ArticleAsync` 方法中针对 `AsyncManager` 的 `OutstandingOperations` 属性的 `Increment` 和 `Decrement` 方法调用注释掉，`ArticleCompleted` 方法是否还能正常运行呢？

```
public class HomeController : AsyncController
{
    public void ArticleAsync(string name)
    {
        //AsyncManager.OutstandingOperations.Increment();
        Task.Factory.StartNew(() =>
        {
            string path = ControllerContext.HttpContext.Server
                .MapPath(string.Format(@"\articles\{0}.html", name));
            using (StreamReader reader = new StreamReader(path))
            {
                AsyncManager.Parameters["content"] = reader.ReadToEnd();
            }
            //AsyncManager.OutstandingOperations.Decrement();
        });
    }

    public ActionResult ArticleCompleted(string content)
    {
        return Content(content);
    }
}
```

实际上在这种情况下 `ArticleCompleted` 依然会被执行，但是如果真的这样做我们就不能确保正常读取文章内容，因为 `ArticleCompleted` 方法会在 `ArticleAsync` 方法执行之后被立即执行。如果文章内容读取是一个相对耗时的操作，`ArticleCompleted` 方法的 `content` 参数（表示读取文章内容的）在执行的时候可能尚未被初始化，那么这种情况下的 `ArticleCompleted` 是如何被执行的呢？

原因很简单，`ReflectedAsyncActionDescriptor` 的 `BeginExecute` 方法在执行 `XxxAsync` 方法的前后会自行调用 `AsyncManager` 的 `OutstandingOperations` 属性的 `Increment` 和 `Decrement` 方法。对于我们给出的例子来说，在执行 `ArticleAsync` 之前 `Increment` 方法被调用使计数器的值变成 1，随后 `ArticleAsync` 被执行，由于该方法以异步的方式读取指定的文件内容，所以会立即返回。最后 `Decrement` 方法被执行使计数器的值变成 0，`AsyncManager` 的 `Completed` 事件被触发并导致 `ArticleCompleted` 方法的执行。而此时文件内容的读取正在进行之中，表示文章内容的 `content` 参数自然尚未被初始化。

`ReflectedAsyncActionDescriptor` 这样的执行机制也对我们使用 `AsyncManager` 提出了要求，那就是对尚未完成的异步操作计数器的增加操作不应该发生在异步线程中。对于如下所示的两种编程方式，第一种是不正确的。

```
//错误
public class HomeController : AsyncController
{
    //其他成员
    public void XxxAsync(string name)
    {
        Task.Factory.StartNew(() =>
        {
            AsyncManager.OutstandingOperations.Increment();
            //...
            AsyncManager.OutstandingOperations.Decrement();
        });
    }
}

//正确
public class HomeController : AsyncController
{
    //其他成员
    public void XxxAsync(string name)
    {
        AsyncManager.OutstandingOperations.Increment();
        Task.Factory.StartNew(() =>
        {
            //...
            AsyncManager.OutstandingOperations.Decrement();
        });
    }
}
```

最后再强调一点，不论是显式调用 `AsyncManager` 的 `Finish` 方法，还是通过调用 `AsyncManager` 的 `OutstandingOperations` 属性的 `Increment/Decrement` 方法使计数器的值变成零，仅仅是让 `XxxCompleted` 方法得以执行，并不能真正阻止已经开始的异步操作的执行。

异步操作的超时控制

异步操作虽然适合那些相对耗时的 I/O 绑定型操作，但是也并不是说对异步操作执行的时间没有限制。异步超时时限通过 `AsyncManager` 的整型属性 `Timeout` 来控制，它表示超时时限的总毫秒数，其默认值为 45000（45 秒）。如果将 `Timeout` 属性设置为 -1，意味着异步操作执行不再具有任何时间的限制。

对于以 `XxxAsync/XxxCompleted` 形式定义的异步 Action 来说，如果 `XxxAsync` 执行之后在规定的超时时限内 `XxxCompleted` 没有得到执行，一个 `TimeoutException` 异常会被抛出来。如果我们以返回类型为 `Task` 的形式定义异步 Action，通过 `Task` 体现的异步操作的执行时间不受 `AsyncManager` 的 `Timeout` 属性的限制。我们通过如下的代码定义了一个 Action 方法 `Data`，它以异步的方式获取相应的数据并作为 Model 呈现在默认的 View 中。由于异步操作中具有一个无限循环，当我们访问该 `Data` 方法时，异步操作将会无限制地执行下去，但是不会有 `TimeoutException` 异常发生。

```
public class HomeController : AsyncController
{
    public Task<ActionResult>Data()
    {
        return Task.Factory.StartNew(() =>
        {
            while (true)
            {}
            return GetModel();

        }).ContinueWith<ActionResult>(task =>
        {
            object model = task.Result;
            return View(task.Result);
        });
    }
}
```

在 ASP.NET MVC 应用编程接口中具有两个特性用于指定异步操作执行的超时时限，它们是具有如下定义的 `System.Web.Mvc.AsyncTimeoutAttribute` 和 `System.Web.Mvc.NoAsyncTimeoutAttribute`。

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
    Inherited=true, AllowMultiple=false)]
public class AsyncTimeoutAttribute : ActionFilterAttribute
{
    public AsyncTimeoutAttribute(int duration);
    public override void OnActionExecuting(ActionExecutingContext filterContext);
    public int Duration { get; }
}

[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
    Inherited=true, AllowMultiple=false)]
public sealed class NoAsyncTimeoutAttribute : AsyncTimeoutAttribute
{
    // Methods
    public NoAsyncTimeoutAttribute() : base(-1)
    {}
}
```

从上面给出的定义可以看出这两个特性均是 `ActionFilter`。`AsyncTimeoutAttribute` 的构造函数接受一个表示超时时限（以毫秒为单位）的整数作为其参数，它通过重写 `OnActionExecuting` 方法将指定的超时时限赋值给 `AsyncManager` 的 `Timeout` 属性。`NoAsyncTimeoutAttribute` 是 `AsyncTimeoutAttribute` 的继承者，它将超时时限设置为-1，意味着它解除了对超时的限制。

从应用在这两个特性的 `AttributeUsageAttribute` 定义可以看出它们既可以应用于类也可以用于方法，意味着我们可以将它们应用到 `Controller` 类上，也可以应用于异步 `Action` 方法（仅对 `XxxAsync` 方法有效，不能应用到 `XxxCompleted` 方法上）上。如果将它们同时应用到 `Controller` 类和 `Action` 方法上，针对方法级别的特性无疑具有更高的优先级。

7.2 Action 方法的执行

Action 方法的执行具有两种基本的形式，即同步执行和异步执行，而在 ASP.NETMVC 的整个体系中涉及到很多同步/异步的执行方式。虽然在前面的章节中已经对此作了相关的介绍，为了让读者对此有一个总体了解，我们在这里来做一个总结性的论述。

7.2.1 MvcHandler 对请求的处理

对于 ASP.NET MVC 应用来说，MvcHandler 是最终用于处理请求的 `HttpHandler`，它是通过 `UrlRoutingModule` 这个实现了 URL 路由的 `HttpModule` 被动态映射到当前请求的。MvcHandler 借助于 `ControllerFactory` 激活并执行目标 `Controller`，并在执行结束后负责对 `Controller` 的释放，相关的内容请参考第 3 章“Controller 的激活”。

如下面的代码片段所示，MvcHandler 同时实现了 `IHttpHandler` 和 `IHttpAsyncHandler` 接口，所以它总是调用 `BeginProcessRequest/EndProcessRequest` 方法以异步的方式来处理请求。

```
public class MvcHandler :
    IHttpAsyncHandler,
    IHttpHandler, ...
{
    //其他成员
    IAsyncResult IHttpAsyncHandler.BeginProcessRequest(HttpContext context,
        AsyncCallback cb, object extraData);
    void IHttpAsyncHandler.EndProcessRequest(IAsyncResult result);
    void IHttpHandler.ProcessRequest(HttpContext httpContext);
}
```

7.2.2 Controller 的执行

通过第 3 章“Controller 的激活”的介绍我们知道 Controller 也具有同步与异步两个版本，它们分别实现了具有如下定义的两个接口 `Controller` 和 `IAsyncController`。激活的 Controller 对象在 MvcHandler 的 `BeginProcessRequest` 方法中是按照这样的方式执行的：如果 Controller 的类型实现了 `IAsyncController` 接口，则调用 `BeginExecute/EndExecute` 方法以异步的方式执行，否则 Controller 是通过调用 `Execute` 方法以同步方式执行的。

```
public interface IController
{
    void Execute(RequestContext requestContext);
}

public interface IAsyncController : IController
{
    IAsyncResult BeginExecute(RequestContext requestContext,
        AsyncCallback callback, object state);
    void EndExecute(IAsyncResult asyncResult);
}
```

默认情况下通过 Visual Studio 的向导创建的 Controller 类型是抽象类型 Controller 的子类。如下面的代码片段所示，它同时实现了 IController 和 IAsyncController 这两个接口，所以 MvcHandler 总是以异步的方式来执行 Controller。

```
public abstract class Controller :
    ControllerBase,
    IController,
    IAsyncController,
    ...
{
    //其他成员
    protected virtual bool DisableAsyncSupport
    {
        get{return false;}
    }
}
```

但是 Controller 类型具有一个受保护的只读属性 DisableAsyncSupport，它用于控制是否禁用对异步执行的支持。在默认情况下，该属性值为 False，所以默认情况下是支持 Controller 的异步执行的。如果我们通过重写该属性将值设置为 True，那么 Controller 将只能以同步的方式执行。具体的实现逻辑体现在如下的代码片段中。BeginExecute 方法在 DisableAsyncSupport 属性为 True 的情况下通过调用 Execute 方法（该方法会调用一个受保护的虚方法 ExecuteCore 最终对 Controller 进行同步执行）同步地执行 Controller，否则通过调用 BeginExecuteCore/EndExecuteCore 以异步方式执行 Controller。

```
public abstract class Controller: ...
{
    //其他成员
    protected virtual IAsyncResult BeginExecute(RequestContext requestContext,
        AsyncCallback callback, object state)
    {
        if (this.DisableAsyncSupport)
        {
            //通过调用 Execute 方法同步执行 Controller
        }
        else
        {
            //通过调用 BeginExecuteCore/EndExecuteCore 方法异步执行 Controller
        }
    }
    protected override void ExecuteCore();
    protected virtual IAsyncResult BeginExecuteCore(AsyncCallback callback,
        object state);
    protected virtual void EndExecuteCore(IAsyncResult asyncResult);
}
```

7.2.3 ActionInvoker 的执行

包括 Model 绑定与验证在内的整个 Action 的执行是通过一个名为 ActionInvoker 的组件来完成的，它同样具有同步和异步两个版本，分别实现了接口 IActionInvoker 和

`IAsyncActionInvoker`。如下面的代码片段所示，这两个接口分别通过 `InvokeAction` 和 `BeginInvokeAction/EndInvokeAction` 方法以同步和异步的方式执行 `Action`。抽象类 `Controller` 中具有一个 `ActionInvoker` 属性用于设置和返回执行自身的 `Action` 的 `ActionInvoker`，该对象最终是通过受保护虚方法 `CreateActionInvoker` 创建的。

```
public interface IActionInvoker
{
    bool InvokeAction(ControllerContext controllerContext, string actionName);
}

public interface IAsyncActionInvoker : IActionInvoker
{
    IAsyncResult BeginInvokeAction(ControllerContext controllerContext,
        string actionName, AsyncCallback callback, object state);
    bool EndInvokeAction(IAsyncResult asyncResult);
}

public abstract class Controller
{
    //其他成员
    public IActionInvoker ActionInvoker { get; set; }
    protected virtual IActionInvoker CreateActionInvoker()
}
```

ASP.NET MVC 真正用于 `Action` 方法同步和异步执行的 `ActionInvoker` 类型分别是 `System.Web.Mvc.ControllerActionInvoker` 和 `System.Web.Mvc.Async.AsyncControllerActionInvoker`。如下面的代码片段所示，`ControllerActionInvoker` 具有一个受保护的方法 `GetControllerDescriptor`，它会根据指定的 `ControllerContext` 获取用于描述当前 `Controller` 的 `ControllerDescriptor` 对象。它的子类 `AsyncControllerActionInvoker` 对这个方法进行了重写。

```
public class ControllerActionInvoker : IActionInvoker
{
    //其他成员
    protected virtual ControllerDescriptor GetControllerDescriptor(
        ControllerContext controllerContext);
}

public class AsyncControllerActionInvoker : ControllerActionInvoker,
    IAsyncActionInvoker, IActionInvoker
{
    //其他成员
    protected override ControllerDescriptor GetControllerDescriptor(
        ControllerContext controllerContext);
}
```

我们需要着重了解的是在默认情况下 `Controller` 采用的 `ActionInvoker` 类型是哪个。ASP.NET MVC 对采用的 `ActionInvoker` 类型的选择机制是这样的。

- 步骤 1：通过当前的 `DependencyResolver` 以 `IAsyncActionInvoker` 接口去获取注册的 `ActionInvoker`，如果返回对象不为 `Null`，则将其作为默认的 `ActionInvoker`，否则进入步骤 2。

- 步骤 2：通过当前的 `DependencyResolver` 以 `IActionInvoker` 接口去获取注册的 `ActionInvoker`，如果返回对象不为 `Null`，则将其作为默认的 `ActionInvoker`，否则进入步骤 3。
- 步骤 3：创建 `AsyncControllerActionInvoker` 对象作为默认的 `ActionInvoker`。

在默认的情况下，当前的 `DependencyResolver` 直接通过对指定的类型进行反射来提供对应的实例对象（`DependencyResolver` 在第 3 章“Controller 的激活”中具有详细介绍），所以前面两个步骤均返回 `Null`，这意味着 Controller 默认使用的 `ActionInvoker` 类型为 `AsyncControllerActionInvoker`，可以通过如下一个简单的实例来验证这一点。

为了验证 `DependencyResolver` 对 `ActionInvoker` 的提供机制的影响，我们在创建的 ASP.NET MVC 应用中将在第 3 章“Controller 的激活”中创建的针对 `Ninject` 的 `NinjectDependencyResolver` 注册为当前的 `DependencyResolver`。然后我们创建了如下两个自定义 `ActionInvoker`，其中 `SyncActionInvoker` 实现了接口 `IActionInvoker`，而 `AsyncActionInvoker` 实现了接口 `IAsyncActionInvoker`。

```
public class SyncActionInvoker : IActionInvoker
{
    //省略成员
}

public class AsyncActionInvoker : IAsyncActionInvoker
{
    //省略成员
}
```

接下来我们定义了如下一个 `HomeController`。方法 `GetActionInvokers` 通过三次调用 `CreateActionInvoker` 方法得到三个具体的 `ActionInvoker` 对象。返回的第一个 `ActionInvoker` 对象代表默认使用的 `ActionInvoker`，其余两个是分别将接口类型 `IActionInvoker` 和 `IAsyncActionInvoker` 注册到当前 `DependencyResolver` 的情况通过调用 `CreateActionInvoker` 方法创建的。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View(this.GetActionInvokers().ToArray());
    }

    public IEnumerable<IActionInvoker> GetActionInvokers()
    {
        NinjectDependencyResolver dependencyResolver
            = (NinjectDependencyResolver)DependencyResolver.Current;
        //1. 默认创建的 ActionInvoker
        yield return this.CreateActionInvoker();

        //2. 为 Dependency 注册针对 IActionInvoker 的类型映射
        dependencyResolver.Register<IActionInvoker, SyncActionInvoker>();
    }
}
```



```

        yield return this.CreateActionInvoker();

        //3. 为 Dependency 注册针对 IAsyncActionInvoker 的类型映射
        dependencyResolver.Register<IAsyncActionInvoker, AsyncActionInvoker>();
        yield return this.CreateActionInvoker();
    }
}

```

在 Action 方法 Index 中, 通过调用 GetActionInvokers 方法得到 ActionInvoker 列表被转换为数组作为 Model 呈现在对应的 View 中。如下所示的是该 View 的定义, 它是一个 Model 类型为 IActionInvoker 数组的强类型 View。在该 View 中我们将每一个 ActionInvoker 的类型通过表格的形式呈现出来。

```

@model IActionInvoker[]
<html>
<head>
    <title>ActionInvoker</title>
</head>
<body>
    <table>
        @for(int i=0; i<Model.Length; i++)
        {
            <tr><td>@(i+1)</td><td>@Model[i].GetType().Name</td></tr>
        }
    </table>
</body>
</html>

```

该程序运行后会在浏览器中呈现如图 7-1 所示的输出结果, 我们前面介绍的创建 ActionInvoker 的三条规则中只有最后一条在这里得到印证。为什么将注册的 ActionInvoker 类型注册在当前 DependencyResolver 对 Controller 的 CreateActionInvoker 方法没有影响呢? (S703)

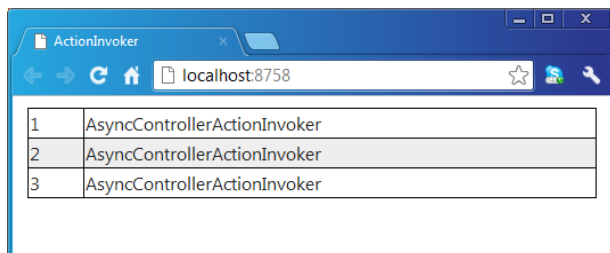


图 7-1 Controller 针对 ActionInvoker 的创建机制 (1)

难道上面介绍的 ActionInvoker 创建机制是错误的吗? 实则不然。造成 Controller 在任何时候总是创建相同类型的 ActionInvoker 的原因在于其内部采用的“缓存”机制。实际上 Controller 的 CreateActionInvoker 方法并没有直接采用通过 DependencyResolver 的静态属性

Current 代表的当前 DependencyResolver，而是使用另一个如下所示的内部静态字段 CurrentCache 代表的 DependencyResolver。

```
public class DependencyResolver
{
    //其他成员
    internal static IDependencyResolver CurrentCache { get; }

    private sealed class CacheDependencyResolver : IDependencyResolver
    {
        // 其他成员
        private readonly ConcurrentDictionary<Type, object> _cache;
    }
}
```

这个 CurrentCache 字段类型为 CacheDependencyResolver，这是一个私有类型，可以看成是对通过静态 Current 属性表示的当前 DependencyResolver 的封装。CacheDependencyResolver 内部使用了被封装的 DependencyResolver 进行对象的激活，但是它会对激活的对象进行缓存，而作为缓存容器的就是通过其只读字段 _cache 表示的 ConcurrentDictionary<Type, object> 对象，所有 Controller 的 CreateActionInvoker 方法返回的都是第一次创建的对象。

为了证实这一点，我们在 HomeController 中定义了如下一个 ClearCachedActionInvokers 方法用于清除当前 CacheDependencyResolver 被缓存的所有对象。GetActionInvokers 方法在调用 CreateActionInvoker 方法之前按照如下的方式及时地调用 ClearCachedActionInvokers 方法将之前缓存的 ActionInvoker 进行了清理。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View(this.GetActionInvokers().ToArray());
    }

    public IEnumerable<IActionInvoker> GetActionInvokers()
    {
        NinjectDependencyResolver dependencyResolver =
            (NinjectDependencyResolver)DependencyResolver.Current;
        yield return this.CreateActionInvoker();

        this.ClearCachedActionInvokers();
        dependencyResolver.Register<IActionInvoker, SyncActionInvoker>();
        yield return this.CreateActionInvoker();

        this.ClearCachedActionInvokers();
        dependencyResolver.Register<IAsyncActionInvoker, AsyncActionInvoker>();
        yield return this.CreateActionInvoker();
    }

    private void ClearCachedActionInvokers()
    {
    }
}
```

```

{
    PropertyInfo property = typeof(DependencyResolver).GetProperty(
        "CurrentCache", BindingFlags.NonPublic | BindingFlags.Static);
    var cachedActionInvoker = property.GetValue(null, null);
    FieldInfo field = cachedActionInvoker.GetType().GetField("_cache",
        BindingFlags.NonPublic | BindingFlags.Instance);
    ConcurrentDictionary<Type, object> dictionary =
        (ConcurrentDictionary<Type, object>)field
            .GetValue(cachedActionInvoker);
    dictionary.Clear();
}
}

```

再次运行我们的程序后将会得到如图 7-2 所示的 `ActionInvoker` 类型列表,这个列表就和我们之前介绍的四条 `ActionInvoker` 创建规则相匹配了。(S704)

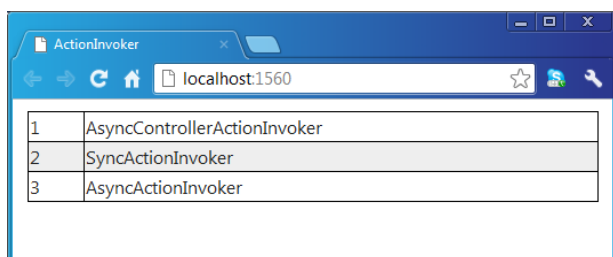


图 7-2 Controller 针对 `ActionInvoker` 的创建机制 (2)

7.2.4 ControllerDescriptor 的同步与异步

如果 Controller 使用 `ControllerActionInvoker`,它所有的 Action 总是以同步的方式来执行,但是当 `AsyncControllerActionInvoker` 作为 Controller 的 `ActionInvoker` 时,却并不意味着总是以异步的方式来执行所有的 Action。

至于这两种类型的 `ActionInvoker` 具体采用怎样的执行方式,涉及到两个描述对象,即用于描述 Controller 和 Action 的 `ControllerDescriptor` 和 `ActionDescriptor`。ASP.NET MVC 具有两个具体的 `ControllerDescriptor`,即 `ReflectedControllerDescriptor` 和 `ReflectedAsyncControllerDescriptor`,它们分别代表同步和异步版本的 `ControllerDescriptor`。

`ReflectedControllerDescriptor` 和 `ReflectedAsyncControllerDescriptor` 并非对实现了 `IController` 和 `IAyncController` 接口的 Controller 的描述(实际上抽象类 Controller 同时实现这两个接口)。两者的区别在于创建者的不同,在默认情况下 `ReflectedControllerDescriptor` 是通过 `ControllerActionInvoker` 创建的,而 `ReflectedAsyncControllerDescriptor` 的创建者则是 `AsyncControllerActionInvoker`。`ActionInvoker` 和 `ControllerDescriptor` 之间的关系可以通过如图 7-3 所示的 UML 来表示。

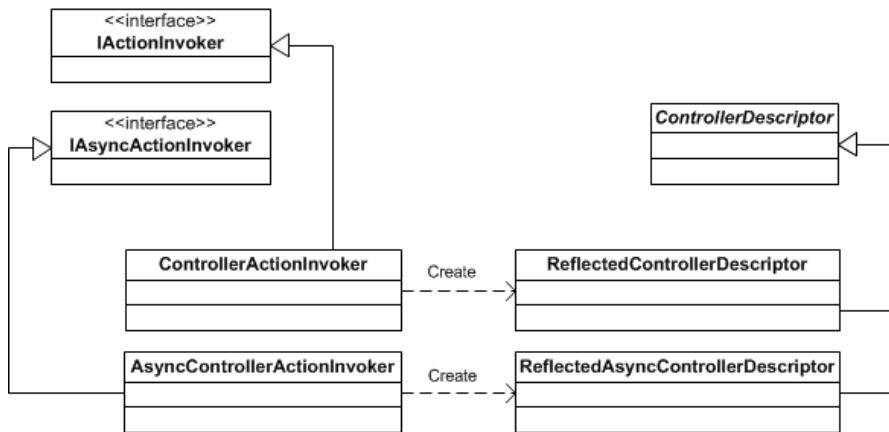


图 7-3 ActionInvoker 与 ControllerDescriptor 之间的关系

ActionInvoker 与 ControllerDescriptor 之间的关系可以通过一个简单的实例来验证。在一个 ASP.NET MVC 应用中，我们创建了如下两个分别继承自 ControllerActionInvoker 和 AsyncControllerActionInvoker 的自定义 ActionInvoker 类型。这两个自定义 ActionInvoker 具有一个公有的 GetControllerDescriptor 方法覆盖了基类的同名方法（受保护的虚方法），并直接返回通过调用基类的同名方法返回的 ControllerDescriptor 对象。

```

public class SyncActionInvoker : ControllerActionInvoker
{
    public new ControllerDescriptor GetControllerDescriptor(
        ControllerContext controllerContext)
    {
        return base.GetControllerDescriptor(controllerContext);
    }
}

public class AsyncActionInvoker : AsyncControllerActionInvoker
{
    public new ControllerDescriptor GetControllerDescriptor(
        ControllerContext controllerContext)
    {
        return base.GetControllerDescriptor(controllerContext);
    }
}
  
```

接下来我们定义了如下两个名为 FooController 和 BarController 的 Controller 类型，在重写的 CreateActionInvoker 方法中分别返回 SyncActionInvoker 和 AsyncActionInvoker 对象。

```

public class FooController : Controller
{
    protected override IActionInvoker CreateActionInvoker()
    {
        return new SyncActionInvoker();
    }
}
  
```

```
public classBarController : Controller
{
    protected override IActionInvoker CreateActionInvoker()
    {
        return new AsyncActionInvoker();
    }
}
```

最后我们定义了如下一个 `HomeController`。`GetControllerDescriptors` 方法返回用于描述指定 `Controller` 的 `ControllerDescriptor` 对象列表。对于每个指定的 `Controller` 对象，我们先获取其 `ActionInvoker` 对象，然后转换成上面定义的 `SyncActionInvoker/AsyncActionInvoker`，并调用其 `GetControllerDescriptor` 方法得到对应的 `ControllerDescriptor` 对象。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View(this.GetControllerDescriptors(
            new FooController(), new BarController()));
    }

    private IEnumerable<ControllerDescriptor> GetControllerDescriptors(
        params Controller[] controllers)
    {
        controllers = controllers ?? new Controller[0];
        foreach (Controller controller in controllers)
        {
            ControllerContext.Controller = controller;
            SyncActionInvoker syncActionInvoker =
                controller.ActionInvoker as SyncActionInvoker;
            AsyncActionInvoker asyncActionInvoker =
                controller.ActionInvoker as AsyncActionInvoker;
            if (null != syncActionInvoker)
            {
                yield return syncActionInvoker.GetControllerDescriptor(
                    ControllerContext);
            }

            if (null != asyncActionInvoker)
            {
                yield return asyncActionInvoker.GetControllerDescriptor(
                    ControllerContext);
            }
        }
    }
}
```

在默认的动作方法 `Index` 中，我们调用 `GetControllerDescriptors` 方法获取用于描述 `FooController` 和 `BarController` 的两个 `ControllerDescriptor` 对象，然后将返回值作为 `Model` 呈现在具有如下定义的 `View` 中。这是一个 `Model` 类型为 `IEnumerable<ControllerDescriptor>` 的强类型 `View`，在该 `View` 中我们将每个 `ControllerDescriptor` 的类型以及对应的 `Controller` 类型通过表格的形式呈现出来。

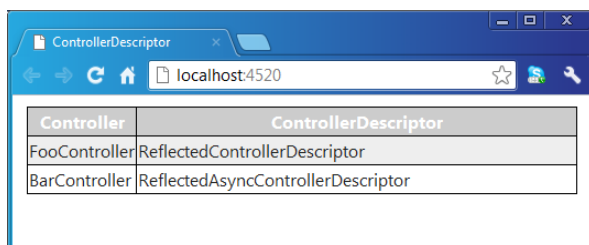
```
@model IEnumerable<ControllerDescriptor>
<html>
```

```

<head>
  <title>ControllerDescriptor</title>
</head>
<body>
  <table>
    <tr><th>Controller</th><th>ControllerDescriptor</th></tr>
    @foreach (ControllerDescriptor descriptor in Model)
    {
      <tr>
        <td>@descriptor.ControllerType.Name</td>
        <td>@descriptor.GetType().Name</td>
      </tr>
    }
  </table>
</body>
</html>

```

该程序运行之后会在浏览器中呈现出如图 7-4 所示的输出结果，可以看到采用 `ControllerActionInvoker` 和 `AsyncControllerActionInvoker` 作为 `ActionInvoker` 的 `FooController` 和 `BarController`，对应的 `ControllerDescriptor` 类型分别为 `ReflectedControllerDescriptor` 和 `ReflectedAsyncControllerDescriptor`。（S705）



Controller	ControllerDescriptor
FooController	ReflectedControllerDescriptor
BarController	ReflectedAsyncControllerDescriptor

图 7-4 ActionInvoker 对 ControllerDescriptor 的创建

7.2.5 ActionDescriptor 的执行

Action 方法可以采用同步和异步执行方式，异步 Action 对应的 `ActionDescriptor` 直接或者间接继承自抽象类 `AsyncActionDescriptor`，后者又是抽象类 `ActionDescriptor` 的子类。如下面的代码片段所示，同步和异步 Action 的执行分别通过调用 `Execute` 和 `BeginExecute/EndExecute` 方法来完成。值得一提的是，`AsyncActionDescriptor` 重写了 `Execute` 方法并直接在此方法中抛出一个 `InvalidOperationException` 异常，意味着 `AsyncActionDescriptor` 对象只能采用异步执行方式。

```

public abstract class ActionDescriptor : ICustomAttributeProvider
{
    //其他成员
    public abstract object Execute(ControllerContext controllerContext,
        IDictionary<string, object> parameters);
}

public abstract class AsyncActionDescriptor : ActionDescriptor
{

```

```

//其他成员
public abstract IAsyncResult BeginExecute(
    ControllerContext controllerContext,
    IDictionary<string, object> parameters, AsyncCallback callback,
    object state);
public abstract object EndExecute(IAsyncResult asyncResult);
}

```

ASP.NET MVC 使用 `ReflectedControllerDescriptor` 对象来描述同步 Action, 而异步 Action 具有两种不同的定义方式, 在 `AsyncController` 中以 `XxxAsync/XxxCompleted` 形式定义的异步 Action 通过 `ReflectedAsyncActionDescriptor` 对象来描述, 返回类型为 `Task` 的异步 Action 则通过 `TaskAsyncActionDescriptor` 对象描述。

`ReflectedControllerDescriptor` 包含的所有 `ActionDescriptor` 类型均为 `ReflectedActionDescriptor`。`ReflectedAsyncControllerDescriptor` 描述的 Controller 可以同时包含同步和异步的 Action 方法, `ActionDescriptor` 的类型取决于 Action 方法的定义方式。`ControllerDescriptor` 与 `ActionDescriptor` 之间的关系可以通过如图 7-5 所示的 UML 来表示。

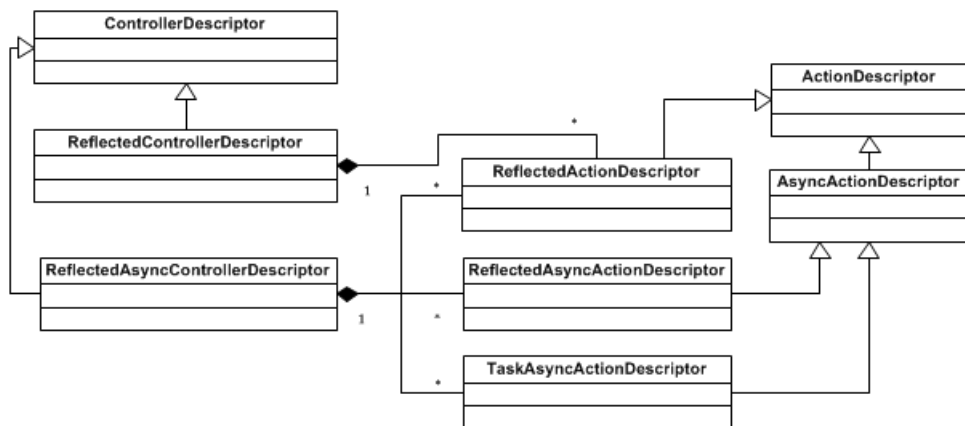


图 7-5 ControllerDescriptor 与 ActionDescriptor 之间的关系

实例演示：ReflectedAsyncControllerDescriptor 中的 ActionDescriptor 类型（S706）

通过 `ControllerActionInvoker` 创建的 `ReflectedControllerDescriptor` 包含的所有 `ActionDescriptor` 类型均为 `ReflectedActionDescriptor`, 而对于通过 `AsyncControllerActionInvoker` 创建的 `ReflectedAsyncControllerDescriptor` 来说, 包含其中的某个 `ActionDescriptor` 的类型取决于对应 Action 方法的定义方式。接下来我们来演示定义在 `AsyncController` 中以不同形式定义的 Action 方法如何决定最终的 `ActionDescriptor` 类型。

我们在 ASP.NET MVC 应用中定义了如下一个 `HomeController`, 它重写的 `CreateActionInvoker` 方法返回一个 `AsyncActionInvoker` 对象。`AsyncActionInvoker` 继承自 `AsyncControllerActionInvoker`, 定义其中的公有 `GetControllerDescriptor` 方法返回描述当前 Controller 的 `ControllerDescriptor` 对象。

```

public class HomeController : AsyncController
{
    protected override IActionInvoker CreateActionInvoker()
    {
        return new AsyncActionInvoker();
    }

    public ActionResult Index()
    {
        AsyncActionInvoker actionInvoker =
            (AsyncActionInvoker)this.ActionInvoker;
        ControllerDescriptor controllerDescriptor =
            actionInvoker.GetControllerDescriptor(ControllerContext);
        List<ActionDescriptor> actionDescriptors = new List<ActionDescriptor>();

        actionDescriptors.Add(controllerDescriptor.FindAction(
            ControllerContext, "Foo"));
        actionDescriptors.Add(controllerDescriptor.FindAction(
            ControllerContext, "Bar"));
        actionDescriptors.Add(controllerDescriptor.FindAction(
            ControllerContext, "Baz"));

        return View(actionDescriptors);
    }

    public void Foo() { }
    public void BarAsync() { }
    public void BarCompleted() { }
    public Task<ActionResult> Baz()
    {
        throw new NotImplementedException();
    }
}

public class AsyncActionInvoker : AsyncControllerActionInvoker
{
    public new ControllerDescriptor GetControllerDescriptor(
        ControllerContext controllerContext)
    {
        return base.GetControllerDescriptor(controllerContext);
    }
}

```

HomeController 定义了三个 Action，其中 Foo 是同步的，Bar 和 Baz 是异步的。两个异步 Action 采用了不同的定义方式，Bar 是通过两个匹配方法 BarAsync/BarCompleted 定义而成，Baz 则直接返回一个 Task<ActionResult>对象。在 Action 方法 Index 中，我们通过 ActionInvoker 得到用于描述这三个 Action 的 ActionDescriptor 对象，并将它们作为 Model 呈现在具有如下所示的 View 中。这是一个 Model 类型为 IEnumerable<ActionDescriptor>的强类型 View，在该 View 中我们将每一个 ActionDescriptor 的类型和它对应的名称通过表格的形式呈现出来。

```

@model IEnumerable<ActionDescriptor>
<html>
<head>

```

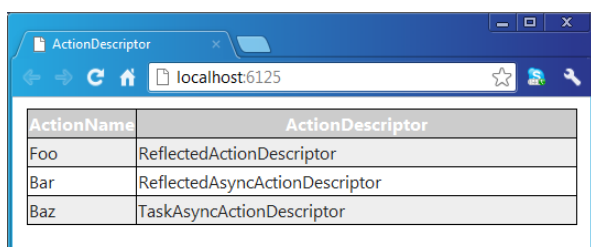


```

        <title>ActionDescriptor</title>
    </head>
    <body>
        <table>
            <tr><th>ActionName</th><th>ActionDescriptor</th></tr>
            @foreach (ActionDescriptor descriptor in Model)
            {
                <tr>
                    <td>@descriptor.ActionName</td>
                    <td>@descriptor.GetType().Name</td>
                </tr>
            }
        </table>
    </body>
</html>

```

该程序运行之后会在浏览器中呈现出如图 7-6 所示的输出结果，可以看出定义在 HomeController 中的三个 Action 由不同类型的 ActionDescriptor 来描述。



ActionName	ActionDescriptor
Foo	ReflectedActionDescriptor
Bar	ReflectedAsyncActionDescriptor
Baz	TaskAsyncActionDescriptor

图 7-6 AsyncController 中具有不同定义方式的 Action 对应的 ActionDescriptor 类型

AsyncController、AsyncControllerActionInvoker 与 AsyncActionDescriptor

以 XxxAsync/XxxCompleted 形式定义异步 Action 必须定义在继承自 AsyncController 的 Controller 类型中，否则将被认为是同步方法。由于通过 ControllerActionInvoker 只能创建包含 ReflectedActionDescriptor 的 ReflectedControllerDescriptor，如果我们在 AsyncController 中采用 ControllerActionInvoker 对象作为 ActionInvoker，所有的 Action 方法也将被认为是同步的。换句话说，真正能够以异步方式执行的 Action 需要同时满足如下两个条件。

- 如果采用 XxxAsync/XxxCompleted 定义方式，对应的 Controller 类型必须继承自 AsyncController，否则只能定义成返回类型为 Task 的方法。
- 所在 Controller 采用 AsyncControllerActionInvoker 来执行 Action。

我们同样可以采用一个简单的实例演示来证实这一点，在 ASP.NET MVC 应用中定义了如下两个 Controller，其中 Controller1 继承自抽象类 Controller，Controller2 则继承自 AsyncController。我们重写了它们的 CreateActionInvoker 方法使 Controller1 返回一个 ControllerActionInvoker 对象，Controller2 返回的则是一个 AsyncControllerActionInvoker 对象（其实默认返回的就是这么一个对象）。两个 Controller 以不同的方式定义了两个异步 Action，其中 Foo 以 FooAsync/FooCompleted 的形式定义，而 Bar 则具有一个 Task<ActionResult> 返回类型。

```

public class Controller1 : Controller
{
    protected override IActionInvoker CreateActionInvoker()
    {
        return new AsyncControllerActionInvoker();
    }

    public void FooAsync()
    { }
    public void FooCompleted()
    { }
    public Task<ActionResult> Bar()
    {
        throw new NotImplementedException();
    }
}

public class Controller2 : AsyncController
{
    protected override IActionInvoker CreateActionInvoker()
    {
        return new ControllerActionInvoker();
    }

    public void FooAsync()
    { }
    public void FooCompleted()
    { }
    public Task<ActionResult> Bar()
    {
        throw new NotImplementedException();
    }
}

```

我们定义了如下一个 HomeController。方法 `GetActionDescriptors` 返回用于描述定义在指定 Controller 中相关 Action 的 `ActionDescriptor` 对象，该方法通过指定 Controller 对象的 `ActionInvoker` 属性获取对应 `ControllerActionInvoker` 或者 `AsyncControllerActionInvoker` 对象，然后以反射的方式调用受保护的 `GetControllerDescriptor` 方法得到用于描述 Controller 的 `ControllerDescriptor` 对象，最终调用 `ControllerDescriptor` 的 `FindAction` 方法得到相关的 `ActionDescriptor` 对象。

```

public class HomeController : AsyncController
{
    public ActionResult Index()
    {
        Dictionary<Type, IEnumerable<ActionDescriptor>> actionDescriptors =
            new Dictionary<Type, IEnumerable<ActionDescriptor>>();
        actionDescriptors.Add(typeof(Controller1), this.GetActionDescriptors(
            new Controller1()));
        actionDescriptors.Add(typeof(Controller2), this.GetActionDescriptors(
            new Controller2()));
        return View(actionDescriptors);
    }

    private IEnumerable<ActionDescriptor> GetActionDescriptors(
        Controller controller)
    {

```

```

{
    ControllerContext.Controller = controller;
    IActionInvoker actionInvoker = controller.ActionInvoker;
    MethodInfo method = actionInvoker.GetType()
        .GetMethod("GetControllerDescriptor",
            BindingFlags.Instance | BindingFlags.NonPublic);
    ControllerDescriptor controllerDescriptor = (ControllerDescriptor)method
        .Invoke(actionInvoker, new object[] { ControllerContext });
    string[] actionNames =
        new string[] { "Foo", "FooAsync", "FooCompleted", "Bar" };
    foreach (string actionName in actionNames)
    {
        ActionDescriptor actionDescriptor = controllerDescriptor.FindAction(
            ControllerContext, actionName);
        if (null != actionDescriptor)
        {
            yield return actionDescriptor;
        }
    }
}
}

```

在 Action 方法 Index 中,我们将创建的 Controller1 和 Controller2 对象作为参数调用 GetActionDescriptors 方法,并将得到的 ActionDescriptor 列表保存到一个 Key 为对应 Controller 类型的字典对象中。最后我们将该字典作为 Model 呈现在具有如下定义的 View 中,在该 View 中我们将 Controller 的类型、Action 的名称和 ActionDescriptor 类型以表格的形式呈现出来。

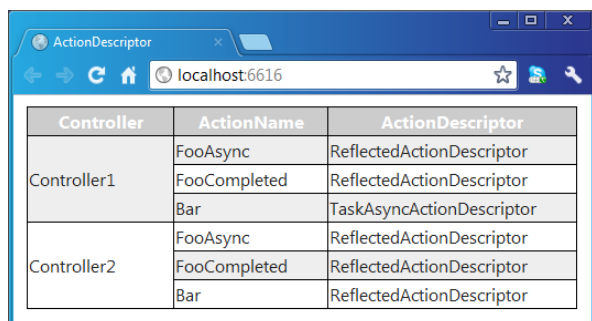
```

@model IDictionary<Type, IEnumerable<ActionDescriptor>>
<html>
<head>
<title>ActionDescriptor</title>
</head>
<body>
<table>
<tr><th>Controller</th><th>ActionName</th><th>ActionDescriptor</th></tr>
@foreach (var item in Model)
{
    ActionDescriptor[] actionDescriptors = item.Value.ToArray();
<tr>
<td rowspan="@item.Value.Count()">@item.Key.Name</td>
<td>@actionDescriptors[0].ActionName</td>
<td>@actionDescriptors[0].GetType().Name</td>
</tr>
for(int i=1; i<actionDescriptors.Length; i++)
{
<tr>
<td>@actionDescriptors[i].ActionName</td>
<td>@actionDescriptors[i].GetType().Name</td>
</tr>
}
}
</table>
</body>
</html>

```

该程序运行之后会在浏览器中呈现出如图 7-7 所示的输出结果,可以看出在 Controller1

和 Controller2 中, FooAsync 和 FooCompleted 方法并未看成是针对一个异步 Action 的定义, 而是作为两个同步 Action 来对待的。对于返回类型为 Task 的 Bar 方法来说, 只有在 Controller1 (采用 AsyncControllerActionInvoker) 中被视为异步 Action, 在 Controller2 (采用 ControllerActionInvoker) 被视为同步 Action。(S707)



Controller	ActionName	ActionDescriptor
Controller1	FooAsync	ReflectedActionDescriptor
	FooCompleted	ReflectedActionDescriptor
	Bar	TaskAsyncActionDescriptor
Controller2	FooAsync	ReflectedActionDescriptor
	FooCompleted	ReflectedActionDescriptor
	Bar	ReflectedActionDescriptor

图 7-7 AsyncController 和 ActionInvoker 对异步 Action 的影响

目标 Action 方法的最终执行由被激活 Controller 的 ActionInvoker 决定, ActionInvoker 通过调用对应的 ActionDescriptor 来执行被它描述的 Action。如果采用 ControllerActionInvoker, 被它创建的 ReflectedControllerDescriptor 只包含同步的 ReflectedActionDescriptor, 所以 Action 方法总是以同步的方式被执行。

对于采用 XxxAsync/XxxCompleted 形式定义的异步 Action, 不论采用怎样的 ActionInvoker, 其 Controller 类型必须是 AsyncController 的子类, 否则被视为两个同步的 Action, 而针对返回类型为 Task 的异步 Action 则无此限制。

7.3 筛选器的执行

在通过 ActionInvoker 对 Action 的执行过程中, ASP.NET MVC 除了利用 ActionDescriptor 执行对应的 Action 方法之外, 还需要执行相关筛选器 (Filter)。ASP.NET MVC 的筛选器是一种基于 AOP (面向方面编程) 的设计, 我们将一些非业务的逻辑实现在相应的筛选器, 并以一种横切 (Crosscutting) 的方式应用到对应的 Action 方法上。在 Action 方法执行前后, 这些筛选器会自动执行。ASP.NET MVC 提供了 AuthorizationFilter、ActionFilter、ResultFilter 和 ExceptionFilter 这四种筛选器, 它们对应着四个接口 IAuthorizationFilter、IActionFilter、IResultFilter 和 IExceptionFilter。

7.3.1 Filter 及其提供机制

ASP.NET MVC 中所谓的筛选器具有两个含义, 一是实现上述四个筛选器接口之一的类型, 二是指具有如下定义的 System.Web.Mvc.Filter 类型。前者一般以自定义特性的形式定义,

并以声明的方式应用到目标 Controller 类型或者 Action 方法上，它们最终都需要转换成相应的 Filter 对象。为了更好地区分这两个概念，本章以下内容提到的“筛选器”表示前者，后者则用“Filter”来表示。Filter 也可以看成是对筛选器的封装，被封装的筛选器通过 Instance 属性返回。

```
public class Filter
{
    public const int DefaultOrder = -1;
    public Filter(object instance, FilterScope scope, int? order);

    public object      Instance { get; protected set; }
    public int         Order { get; protected set; }
    public FilterScope Scope { get; protected set; }
}

public enum FilterScope
{
    Action      = 30,
    Controller  = 20,
    First       = 0,
    Global      = 10,
    Last       = 100
}
```

多个同类（这里主要指四种筛选器类型）的筛选器可以应用到同一个 Action 方法上，它们执行的顺序通过 Order 和 Scope 属性来决定。Order 属性对应数值越小，执行的优先级越高。该属性的默认值为-1，对应着 Filter 中定义的常量 DefaultOrder。

如果两个 Filter 具有相同的 Order 属性值，那么 Scope 属性最终决定谁被优先执行。Filter 的 Scope 属性返回一个类型为 System.Web.Mvc.FilterScope 的枚举，该枚举表示应用 Filter 的范围。枚举项 Action 和 Controller 代表 Action 方法和 Controller 类级别，First 和 Last 意味着希望被作为第一个和最后一个 Filter 来执行，而 Global 代表一个全局的 Filter。

通过上面的代码片段可以看到，FilterScope 的 5 个枚举选项均被设置了一个对应的数值，这个值决定了 Filter 的执行顺序，具有更小的枚举值会被优先执行。从 FilterScope 的定义可以得到这样的结论：对于具有相同 Order 属性值的多个 Filter，应用在 Controller 上的 Filter 比应用在 Action 方法上的 Filter 具有更高的执行优先级，一个全局的 Filter 的执行优先级又高于基于 Controller 的 Filter。

FilterProvider

Filter 的提供机制与我们之前介绍的基于 ModelMetadata、ModelBinder 和 ModelValidator 的提供机制类似，均是通过相应的 Provider 来提供的。提供筛选器的 FilterProvider 实现了具有如下定义的接口 System.Web.Mvc.IFilterProvider，该接口定义了唯一的方法 GetFilters，它根据指定的 ControllerContext 和用于描述目标 Action 的 ActionDescriptor 获取所有的 Filter 列表。

```
public interface IFilterProvider
{
    IEnumerable<Filter> GetFilters(ControllerContext controllerContext,
        ActionDescriptor actionDescriptor);
}
```

可以通过静态类型 `System.Web.Mvc.FilterProviders` 注册和获取当前使用的 `FilterProvider`。如下面的代码片段所示，`FilterProviders` 具有一个类型为 `System.Web.Mvc.FilterProviderCollection` 的只读属性 `Providers`，表示当前使用的 `FilterProvider` 列表。`FilterProviderCollection` 是元素类型为 `IFilterProvider` 的集合，其 `GetFilters` 方法用于获取集合中所有 `FilterProvider` 对象提供的 `Filter`。

```
public static class FilterProviders
{
    public static FilterProviderCollection Providers { get; }
}

public class FilterProviderCollection : Collection<IFilterProvider>
{
    //其他成员
    public IEnumerable<Filter> GetFilters(ControllerContext controllerContext,
        ActionDescriptor actionDescriptor);
}
```

ASP.NET MVC 提供了三种原生的 `FilterProvider`，分别是 `FilterAttributeFilterProvider`、`ControllerInstanceFilterProvider` 和 `GlobalFilterCollection`，接下来就对它们进行单独的介绍。

FilterAttribute 与 FilterAttributeFilterProvider

我们通常将筛选器定义成特性并以声明的方式应用到 `Controller` 类型或者 `Action` 方法上，而抽象类型 `System.Web.Mvc.FilterAttribute` 是所有筛选器特性的基类。如下面的代码片段所示，`FilterAttribute` 特性实现了 `System.Web.Mvc.IMvcFilter` 接口，实现的两个只读属性 `Order` 和 `AllowMultiple` 分别用于控制筛选器的执行顺序以及判断多个相同类型的筛选器特性能否同时应用到同一个目标元素（类或者方法）。

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, Inherited=true,
    AllowMultiple=false)]
public abstract class FilterAttribute : Attribute, IMvcFilter
{
    protected FilterAttribute();

    public bool AllowMultiple { get; }
    public int Order { get; set; }
}

public interface IMvcFilter
{
    bool AllowMultiple { get; }
    int Order { get; }
}
```

从应用在 `FilterAttribute` 上的 `AttributeUsageAttribute` 特性的定义可以看出该特性可以应用在类型和方法上,这意味着筛选器一般都可以应用在 `Controller` 类型和 `Action` 方法上。只读属性 `AllowMultiple` 实际上返回的是 `AttributeUsageAttribute` 特性的同名属性,通过上面的定义可以看到默认情况下该属性值为 `False`。

分别用于描述 `Controller` 和 `Action` 的类型 `ControllerDescriptor` 和 `ActionDescriptor` 均实现了 `ICustomAttributeProvider` 接口,可以调用相应的方法获取应用在对应的 `Controller` 类型或者 `Action` 方法上包括 `FilterAttribute` 在内的所有特性。实际上这两个类型提供了单独的方法 `GetFilterAttributes` 获取 `FilterAttribute` 特性列表。如下面的代码片段所示,该方法具有一个布尔类型的参数 `useCache`,表示是否需要解析出来的 `FilterAttribute` 特性进行缓存以缓解频繁的反射操作对性能造成的影响。

```
public abstract class ControllerDescriptor : ICustomAttributeProvider,
    IUniquelyIdentifiable
{
    //其他成员
    public virtual IEnumerable<FilterAttribute> GetFilterAttributes(
        bool useCache);
}
public abstract class ActionDescriptor : ICustomAttributeProvider,
    IUniquelyIdentifiable
{
    //其他成员
    public virtual IEnumerable<FilterAttribute> GetFilterAttributes(
        bool useCache);
}
```

针对 `FilterAttribute` 的 `Filter` 通过具有如下定义的 `System.Web.Mvc.FilterAttributeFilterProvider` 对象来提供,它直接调用当前 `ControllerDescriptor` 和 `ActionDescriptor` 的 `GetFilterAttributes` 方法获取所有应用在 `Controller` 类型和当前 `Action` 方法的 `FilterAttribute` 特性,并借此创建相应的 `Filter` 对象。`FilterAttributeFilterProvider` 构造函数的参数 `cacheAttributeInstances` 表示是否启用针对 `FilterAttribute` 的缓存,它将作为调用 `GetFilterAttributes` 方法的参数。在默认的情况下(通过调用默认无参的构造函数创建的 `FilterAttributeFilterProvider`) `FilterAttributeFilterProvider` 是支持缓存的。

```
public class FilterAttributeFilterProvider : IFilterProvider
{
    public FilterAttributeFilterProvider();
    public FilterAttributeFilterProvider(bool cacheAttributeInstances);

    protected virtual IEnumerable<FilterAttribute> GetActionAttributes(
        ControllerContext controllerContext, ActionDescriptor actionDescriptor);
    protected virtual IEnumerable<FilterAttribute> GetControllerAttributes(
        ControllerContext controllerContext, ActionDescriptor actionDescriptor);
    public virtual IEnumerable<Filter> GetFilters(
        ControllerContext controllerContext, ActionDescriptor actionDescriptor);
}
```

对于通过调用 `GetFilters` 得到的每个 `Filter`,对应的 `FilterAttribute` 特性作为其 `Instance` 属性。`Order` 属性来源于 `FilterAttribute` 的同名属性,而 `Scope` 属性则取决于 `FilterAttribute` 特性

是应用在 Controller 类型上（Scope 属性值为 Controller）还是当前的 Action 方法上（Scope 属性值为 Action）。

Controller 与 ControllerInstanceFilterProvider

提到 ASP.NET MVC 的筛选器，大部分读者都只会想到各种 FilterAttribute 特性，实际上 Controller 本身就是一个筛选器。如下面的代码片段所示，抽象类 Controller 实现了 IActionFilter、IAuthorizationFilter、IExceptionFilter 和 IResultFilter 四个接口。

```
public abstract class Controller : ControllerBase,
    IActionFilter,
    IAuthorizationFilter,
    IExceptionFilter,
    IResultFilter,
    ...
{
    //省略成员
}
```

ASP.NET MVC 通过具有如下定义的 System.Web.Mvc.ControllerInstanceFilterProvider 类型来表示针对 Controller 对象这种特殊筛选器的 Filter。它的 GetFilters 方法根据指定的 ControllerContext 获取对应的 Controller 对象，并以此创建一个对应的 Filter（Controller 对象作为 Filter 对象的 Instance 属性值）。值得注意的是这个 Filter 的 Scope 不是 Controller 而是 First，Order 的值为-2147483648（Int32.MinValue），毫无疑问这样的 Filter 肯定被第一个执行。

```
public class ControllerInstanceFilterProvider : IFilterProvider
{
    public IEnumerable<Filter> GetFilters(ControllerContext controllerContext,
        ActionDescriptor actionDescriptor);
}
```

GlobalFilterCollection

FilterAttribute 需要显式地应用到某个具体的 Controller 类型或者 Action 方法上，但是在有些情况下需要一种全局性的 Filter。所谓全局 Filter，就是不需要显式与某个 Controller 或者 Action 进行关联就会默认应用到定义在所有 Controller 中的 Action 方法上，这种全局 Filter 通过具有如下定义的 System.Web.Mvc.GlobalFilterCollection 来提供。

```
public sealed class GlobalFilterCollection : IEnumerable<Filter>,
    IEnumerable, IFilterProvider
{
    public GlobalFilterCollection();
    public void Add(object filter);
    public void Add(object filter, int order);
    private void AddInternal(object filter, int? order);
    public void Clear();
    public bool Contains(object filter);
    public IEnumerator<Filter> GetEnumerator();
    public void Remove(object filter);
}
```



```

IEnumerator IEnumerable.GetEnumerator();
IEnumerable<Filter> IFilterProvider.GetFilters(
    ControllerContext controllerContext, ActionDescriptor actionDescriptor);

public int Count { get; }
}

```

`GlobalFilterCollection` 是一个元素类型为 `Filter` 的集合，同时自身也是实现了 `IFilterProvider` 接口的 `FilterProvider`，它实现的 `GetFilters` 方法返回的就是它自己。通过 `GlobalFilterCollection` 提供的方法我们可以实现对全局 `Filter` 的添加、删除和清除。用于添加 `Filter` 的 `Add` 方法的参数 `filter` 不是一个 `Filter` 对象而是一个具体筛选器对象。ASP.NET MVC 根据指定的筛选器创建对应的 `Filter`（指定的筛选器作为 `Filter` 的 `Instance` 属性）并添加到集合之中，这个 `Filter` 对象的 `Scope` 属性被设置成 `Global`。

在调用 `Add` 方法进行全局 `Filter` 注册的时候，我们可以指定相应的 `Order` 属性，如果没有显式指定并且指定的筛选器是一个 `FilterAttribute` 对象，那么该特性的 `Order` 属性将会作为 `Filter` 对象的 `Order`，否则添加的 `Filter` 对象的 `Order` 属性被设置成默认值-1。

针对整个 Web 应用的全局 `Filter` 的注册和获取可以通过静态类型 `System.Web.Mvc.GlobalFilters` 来实现。如下面的代码片段所示，`GlobalFilters` 具有一个静态只读属性 `Filters` 返回一个 `GlobalFilterCollection` 对象，它就代表这个全局的 `Filter` 列表。

```

public static class GlobalFilters
{
    public static GlobalFilterCollection Filters { get; }
}

```

到目前为止，我们已经介绍了 ASP.NET MVC 默认提供的三种 `FilterProvider`，以及各自采用的 `Filter` 提供机制。当用于注册 `FilterProvider` 的静态类型 `FilterProviders` 被加载的时候，它会默认创建这三种类型的对象并添加到通过静态 `Providers` 属性表示的 `FilterProvider` 列表中，具体的逻辑体现在如下所示的代码片段中。也就是说，ASP.NET MVC 在默认的情况下会采用这三种 `FilterProvider` 来提供所有的 `Filter` 对象。

```

public static class FilterProviders
{
    static FilterProviders()
    {
        Providers = new FilterProviderCollection();

        Providers.Add(GlobalFilters.Filters);
        Providers.Add(new FilterAttributeFilterProvider());
        Providers.Add(new ControllerInstanceFilterProvider());
    }

    public static FilterProviderCollection Providers { get; }
}

```

实例演示：验证 `Filter` 的提供机制和执行顺序（S708，S709，S710）

为了让读者对上面介绍的三种 `FilterProvider` 和各自实现的 `Filter` 提供机制具有一个更加深刻的认识，我们来做一个简单的实例演示。在一个 ASP.NET MVC 应用中定义了

FooAttribute、BarAttribute 和 BazAttribute 三个 ActionFilter，如下面的代码片段所示，它们都继承自我们自定义的 FilterBaseAttribute。

```
public abstract class FilterBaseAttribute:FilterAttribute, IActionFilter
{
    public void OnActionExecuted(ActionExecutedContext filterContext)
    {}

    public void OnActionExecuting(ActionExecutingContext filterContext)
    {}
}

public class FooAttribute : FilterBaseAttribute
{}
public class BarAttribute : FilterBaseAttribute
{}
public class BazAttribute : FilterBaseAttribute
{}

```

首先在默认生成的 FilterConfig 类型中通过如下的方式将 BazAttribute 注册为一个全局 Filter。由于 FilterConfig 默认情况下会注册一个针对 HandleErrorAttribute 特性的全局 Filter，在这里我们将这行代码注释掉。

```
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        //filters.Add(new HandleErrorAttribute());
        GlobalFilters.Filters.Add(new BazAttribute());
    }
}

```

接下来定义了如下一个 HomeController。我们定义的 FooAttribute 应用在 HomeController 类型上，Action 方法 DemoAction 上则应用了 BarAttribute。在 Action 方法 Index 中我们创建了一个描述 Action 方法 DemoAction 的 ActionDescriptor 对象，并利用静态类型 FilterProviders 得到最终应用到该 Action 的所有 Filter 对象，最后将这个 Filter 集合作为 Model 呈现在默认的 View 中。

```
[Foo]
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ReflectedControllerDescriptor controllerDescriptor =
            new ReflectedControllerDescriptor(typeof(HomeController));
        ActionDescriptor actionDescriptor = controllerDescriptor.FindAction(
            ControllerContext, "DemoAction");
        IEnumerable<Filter> filters = FilterProviders.Providers.GetFilters(
            ControllerContext, actionDescriptor);
        return View(filters);
    }

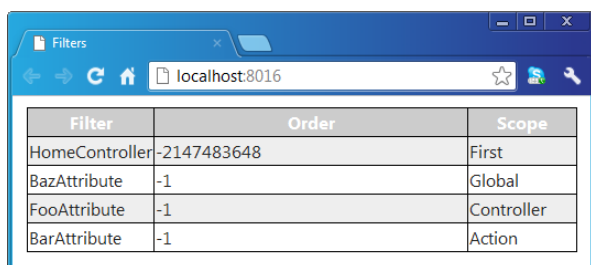
    [Bar]
    public void DemoAction()
    { }
}

```

如下所示的 Action 方法 Index 对应 View 的定义,这是一个 Model 类型为 IEnumerable<Filter> 的强类型 View,在该 View 中我们将所有 Filter 对象的 Instance 属性类型、Order 和 Scope 属性呈现在一个表格中。

```
@model IEnumerable<Filter>
<html>
  <head>
    <title>Filters</title>
  </head>
  <body>
    <table>
      <tr><th>Instance</th><th>Order</th><th>Scope</th></tr>
      @foreach (Filter filter in Model)
      {
        <tr>
          <td>@filter.Instance.GetType().Name</td>
          <td>@filter.Order</td>
          <td>@filter.Scope</td>
        </tr>
      }
    </table>
  </body>
</html>
```

运行程序之后会在浏览器中呈现如图 7-8 所示的输出结果,可以清楚地看到最终应用到 Action 方法 DemoAction 上的筛选器有 4 个,分别对应着应用到 HomeController 类型上的 FooAttribute,应用到 DemoAction 方法上的 BarAttribute,全局注册的 BazAttribute 和 HomeController 本身。(S708)



Filter	Order	Scope
HomeController	-2147483648	First
BazAttribute	-1	Global
FooAttribute	-1	Controller
BarAttribute	-1	Action

图 7-8 通过不同方式注册的 Filter 列表

在前面的内容中我们提到如果同一 Action 具有多个相同类型的 Filter,其 Order 和 Scope 属性最终决定了 Filter 执行的顺序。根据图 7-8 所示的四个 Filter 的 Order/Scope 属性值,它们执行的先后顺序应该是 HomeController=>BazAttribute=>FooAttribute=>BarAttribute,现在就通过实例来证实这一点,为此我们在 FilterBaseAttribute 的 OnActionExecuting 方法中将当前 FilterAttribute 的类型和方法名(“OnActionExecuting”)写入当前的 HttpResponse 并最终呈现在浏览器上。

```
public abstract class FilterBaseAttribute:FilterAttribute, IActionFilter
{
```

```

public void OnActionExecuted(ActionExecutedContext filterContext)
{

}

public void OnActionExecuting(ActionExecutingContext filterContext)
{
    filterContext.HttpContext.Response.Write(
        string.Format("{0}.OnActionExecuting()<br/>", this.GetType()));
}
}

```

然后我们按照相同的方式重写了 HomeController 的 OnActionExecuting 方法，将自身的类型和当前方法名称呈现出来。

```

[Foo]
public class HomeController : Controller
{
    //其他成员
    protected override void OnActionExecuting(
        ActionExecutingContext filterContext)
    {
        Response.Write("HomeController.OnActionExecuting()<br/>");
    }

    [Bar]
    public void DemoAction()
    { }
}

```

再次运行我们的程序并在浏览器上指定正确的地址访问定义在 HomeController 的 Action 方法 DemoAction，会在浏览器中呈现如图 7-9 所示的输出结果。这个结果体现了应用到 Action 方法 Data 上的四个 ActionFilter 执行的顺序，而这与通过 Order 和 Scope 属性决定的顺序是一致的。(S709)

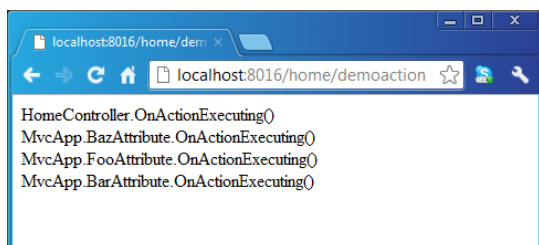


图 7-9 应用在同一个 Action 方法中的同类筛选器的执行顺序

关于 Filter 的提供机制还有另一个值得深究的问题，即我们在定义 FilterAttribute 的时候可以将应用的 AttributeUsageAttribute 特性的 AllowMultiple 属性设置为 False 使它只能在一个目标元素上使用一次，但是我们依然可以在 Action 方法和所在的 Controller 类型上应用它们，甚至可以将它们注册为全局 Filter，那么这些 FilterAttribute 都将有效吗？

现在就来通过实例来验证这一点。删除现有的所有的 FilterAttribute，并定义如下一个类型为 FooAttribute 的 ActionFilter，并将应用在它上面的 AttributeUsageAttribute 特性的

AllowMultiple 属性设置为 False。

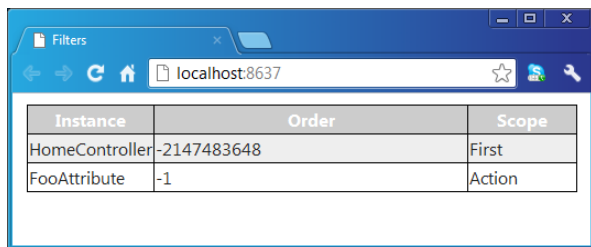
```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
    AllowMultiple = false)]
public class FooAttribute : FilterAttribute, IActionFilter
{
    public void OnActionExecuted(ActionExecutedContext filterContext)
    { }
    public void OnActionExecuting(ActionExecutingContext filterContext)
    { }
}
```

现在我们将该 FooAttribute 特性同时应用在 HomeController 类型和 Action 方法 DemoAction 上，然后在 Global.asax 中注册一个针对它的全局 Filter。

```
[Foo]
public class HomeController : Controller
{
    //其他成员
    [Foo]
    public void DemoAction()
    { }
}

public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        //filters.Add(new HandleErrorAttribute());
        GlobalFilters.Filters.Add(new FooAttribute());
    }
}
```

直接运行我们的程序后浏览器中会呈现出如图 7-10 所示的输出结果，可以清楚地看到虽然我们在三个地方注册了 FooAttribute，但是由于该特性的 AllowMultiple 属性为 False，所以只有其中一个 FooAttribute 最终是有效的。从设计上这也很好理解，虽然我们可以将多个同类的 Filter 注册为不同的范围（Action、Controller 和 Global），但是它们最终还是应用到具体某个 Action 方法上，我们将应用在它们上的 AttributeUsageAttribute 的 AllowMultiple 属性设置为 False，就是希望 Action 方法在执行的时候只有一个 Filter 被执行。（S710）



Instance	Order	Scope
HomeController	-2147483648	First
FooAttribute	-1	Action

图 7-10 AllowMultiple 为 False 的 FilterAttribute 的唯一性

如果以不同的 Scope 注册了多个 AllowMultiple 属性为 False 的 FilterAttribute，最终有效

的是哪个呢？从图 7-10 可以看出，貌似应用在 Action 方法(Scope 为 Action)上的 FooAttribute 是有效的。其实具体的逻辑是这样的：所有被创建的 Filter 按照 Order 和 Scope 进行排序（即 Filter 执行的顺序），最后的那个会被选用。

FilterInfo

ASP.NET MVC 的四种筛选器最终被封装成相应的 Filter 对象，但是它们执行的时机和方式是不同的，所以在执行之前需要根据被封装的筛选器类型对所有 Filter 进行分组。具体来说，当 ActionInvoker 被调用的时候，它会利用静态类型 FilterProviders 得到所有注册的 FilterProvider，并利用它们根据当前 ControllerContext 和描述目标 Action 的 ActionDescriptor 对象得到所有的 Filter 对象，然后根据其 Instance 属性表示的筛选器类型将它们分组，最终得到一个具有如下定义的 System.Web.Mvc.FilterInfo 类型的对象。

```
public class FilterInfo
{
    public FilterInfo();
    public FilterInfo(IEnumerable<Filter> filters);

    public IList<IActionFilter>      ActionFilters { get; }
    public IList<IAuthorizationFilter> AuthorizationFilters { get; }
    public IList<IExceptionFilter>   ExceptionFilters { get; }
    public IList<IResultFilter>      ResultFilters { get; }
}
```

如上面的代码片段所示，可以通过指定一个 Filter 列表来创建一个 FilterInfo 对象，它具有针对 4 种筛选器类型的列表属性，如果我们在构造函数中指定了一个 Filter 列表，那么每个 Filter 的 Instance 属性表示筛选器会被提取出来并根据类型添加到某个对应的筛选器列表之中。

7.3.2 AuthorizationFilter

在总体讲述了筛选器及其提供机制之后，我们按照执行的先后顺序对四种不同的筛选器进行单独介绍，首先来介绍最先执行的 AuthorizationFilter。从命名来看，AuthorizationFilter 用于完成授权相关的工作，所以它应该在 Action 方法被调用之前执行才能起到授权的作用。不仅限于授权，如果我们在目标 Action 方法被调用之前“做点什么”，都可以通过自定义的 AuthorizationFilter 来实现。

AuthorizationFilter 实现了接口 System.Web.Mvc.IAuthorizationFilter。如下面的代码片段所示，IAuthorizationFilter 定义了一个 OnAuthorization 方法用于实现授权的操作。该方法的参数 filterContext 表示一个表示授权上下文的 System.Web.Mvc.AuthorizationContext 对象，它直接继承自 ControllerContext。

```
public interface IAuthorizationFilter
{
    void OnAuthorization(AuthorizationContext filterContext);
}
```

```

public class AuthorizationContext : ControllerContext
{
    public AuthorizationContext();
    public AuthorizationContext(ControllerContext controllerContext,
        ActionDescriptor actionDescriptor);

    public virtual ActionDescriptor ActionDescriptor { get; set; }
    public ActionResult Result { get; set; }
}

```

AuthorizationContext 的 ActionDescriptor 属性表示描述当前执行 Action 的 ActionDescriptor 对象，而 Result 属性返回一个用于在授权阶段直接对请求进行响应的 ActionResult。AuthorizationFilter 的执行是 ActionInvoker 进行 Action 执行的第一项工作，因为后续的工作（Model 绑定、Model 验证、Action 方法执行等）的执行只有在成功授权的情况下才有意义。

ActionInvoker 在执行 AuthorizationFilter 之前，会先根据当前的 ControllerContext 和描述当前 Action 的 ActionDescriptor 创建一个表示授权上下文的 AuthorizationContext 对象。然后它将此 AuthorizationContext 对象作为参数，按照 Filter 对象 Order 和 Scope 属性决定的顺序执行所有 AuthorizationFilter 的 OnAuthorization 方法。

在所有的 AuthorizationFilter 都执行完毕之后，如果 AuthorizationContext 对象的 Result 属性表示的 ActionResult 不为 Null，整个 Action 的执行将会终止，该 ActionResult 将直接被执行用于完成对当前请求的响应。一般来说，某个 AuthorizationFilter 在对当前请求实施授权的时候，如果授权失败它可以通过设置 AuthorizationContext 对象的 Result 属性回复一个“401, Unauthorized”响应，或者重定向到一个错误页面。

AuthorizeAttribute

如果我们要求某个 Action 只能被认证的用户访问，可以在 Controller 类型或者 Action 方法上应用具有如下定义的 System.Web.Mvc.AuthorizeAttribute 特性。AuthorizeAttribute 还可以设置目标 Action 可被访问的用户账号或者角色，字符串属性 Users 和 Roles 用于指定被授权的用户名和角色列表，每个用户名/角色之间用采用逗号作为分隔符。如果没有显式地对 Users 和 Roles 属性进行设置，AuthorizeAttribute 在进行授权操作的时候只要求当前用户是经过认证的。

```

[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
    Inherited=true, AllowMultiple=true)]
public class AuthorizeAttribute : FilterAttribute, IAuthorizationFilter
{
    //其他成员
    public virtual void OnAuthorization(AuthorizationContext filterContext);
    protected virtual HttpValidationStatus OnCacheAuthorization(
        HttpContextBase httpContext);

    public string Roles { get; set; }
    public override object TypeId { get; }
    public string Users { get; set; }
}

```

如果授权失败（当前访问者是未被认证用户，或者当前用户的用户名或者拥有的角色没有指定的授权用户或者角色列表中），一个 `System.Web.Mvc.HttpUnauthorizedResult` 对象会被创建并赋值给 `AuthorizationContext` 的 `Result` 属性，意味着客户端会接收到一个状态为“401, Unauthorized”的响应。

有一些读者会将 `AuthorizeAttribute` 特性对方法的授权与 `PrincipalPermissionAttribute` 特性等同起来，实际上不但它们实现授权的机制不一样（后者是通过代码访问安全检验实现对方调用方法的授权），其授权策略也不一样。以下面定义的两个方法为例，应用了 `PrincipalPermissionAttribute` 特性的 `FooOrAdmin` 方法可以被账号为“Foo”或者拥有“Admin”角色的用户访问，而应用了 `AuthorizeAttribute` 特性的方法 `FooAndAdmin` 方法只能被拥有角色“Admin”的用户“Foo”调用。也就是说 `PrincipalPermissionAttribute` 特性对 `User` 和 `Role` 的授权逻辑是“逻辑或”，而 `AuthorizeAttribute` 采用的则是“逻辑与”。

```
[PrincipalPermission( SecurityAction.Demand,Name="Foo", Role="Admin")]
public void FooOrAdmin()
{ }

[Authorize(Users="Foo", Roles="Admin")]
public void FooAndAdmin()
{ }
```

除此之外，我们可以将多个 `PrincipalPermissionAttribute` 和 `AuthorizeAttribute` 特性应用到同一个类型或者方法上。对于前者，如果当前用户通过了任意一个 `PrincipalPermissionAttribute` 特性的授权就有权调用目标方法，对于后者来说，则意味着需要通过所有 `AuthorizeAttribute` 特性的检验才具有了调用目标方法的权限。以如下两个方法为例，用户“Foo”或者“Bar”可以有限调用 `FooOrBar` 方法，但是没有任何一个用户有权调用 `CannotCall` 方法。

```
[PrincipalPermission( SecurityAction.Demand, Name="Foo")
[PrincipalPermission( SecurityAction.Demand, Name="Bar")]
public void FooOrBar()
{ }

[Authorize(Users="Foo")]
[Authorize(Users="Bar")]
public void CannotCall()
{ }
```

RequireHttpsAttribute

从名称也可以看出来，`System.Web.Mvc.RequireHttpsAttribute` 这个 `AuthorizationFilter` 要求用户总是以 HTTPS 请求的方式访问目标 Action。如果当前并不是一个 HTTPS 请求（通过当前 `HttpRequest` 的 `IsSecureConnection` 属性判断），`RequireHttpsAttribute` 在 HTTP 方法为 GET 的情况下会创建一个 `System.Web.Mvc.RedirectResult` 对象并作为 `AuthorizationContext` 的 `Result` 属性。

`RedirectResult` 用于客户端的重定向，原请求地址的网络协议前缀（Scheme）替换成“https://”

后得到的 URL 作为重定向的目标地址。比如当前请求地址为 `http://www.artech.com/home/index`，`RequireHttpsAttribute` 将其重定向到 `https://www.artech.com/ home/index`。我们将在第 8 章“View 的呈现”中对 `RedirectResult` 进行单独介绍。

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
    Inherited=true, AllowMultiple=false)]
public class RequireHttpsAttribute : FilterAttribute, IAuthorizationFilter
{
    protected virtual void HandleNonHttpRequest(
        AuthorizationContext filterContext);
    public virtual void OnAuthorization(AuthorizationContext filterContext);
}
```

`RequireHttpsAttribute` 特性针对 HTTPS 的重定向仅限于 HTTP-GET 请求，如果当前请求的 HTTP 方法并不是 GET，`RequireHttpsAttribute` 会直接抛出一个 `InvalidOperationException` 异常。如上面的代码片段所示，针对非 HTTPS 请求的处理通过调用受保护的方法 `HandleNonHttpRequest` 来完成，如果我们需要采用不同的处理方法，可以继承 `RequireHttpsAttribute` 并重写该方法。

ValidateInputAttribute

为了避免用户在请求中嵌入一些不合法的内容对网站进行恶意攻击（比如 XSS 攻击），ASP.NET 需要对请求的输入进行验证。如下面的代码片段所示，表示 HTTP 请求的抽象类型 `HttpRequestBase` 具有一个 `ValidateInput` 方法用于验证请求的输入。实际上这个方法仅仅是在请求上作一下标记而已，在读取相应的请求输入时才根据这些标记决定是否需要进行相应的验证，不过为了便于表达，我们还是将针对 `ValidateInput` 方法的调用说成是对请求输入的验证。

```
public abstract class HttpRequestBase
{
    //其他成员
    public virtual void ValidateInput();
}
```

所有 `Controller` 的基类 `ControllerBase` 具有如下一个布尔类型的属性 `ValidateRequest`，它表示是否需要对请求输入进行验证，在默认情况下该属性的默认值为 `True`，意味着针对请求输入的验证在默认情况下是开启的。当 `ActionInvoker` 在完成了对所有 `AuthorizationFilter` 的执行之后，会根据该属性决定是否需要通过调用当前 `HttpRequest` 的 `ValidateInput` 方法进行请求输入的验证。

```
public abstract class ControllerBase : IController
{
    //其他成员
    public bool ValidateRequest { get; set; }
}
```

也正是由于 `ActionInvoker` 针对请求输入验证是在所有 `AuthorizationFilter` 执行之后进行

的,所以我们可以通过自定义 `AuthorizationFilter` 的方式来设置当前 `Controller` 的 `ValidateRequest` 属性进而开启或者关闭针对请求输入的验证。`System.Web.Mvc.ValidateInputAttribute` 就是这么做的,这可以从如下表示 `ValidateInputAttribute` 的定义看出来(构造函数的参数 `enableValidation` 表示是否启动针对请求的输入验证)。

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, Inherited=true,
    AllowMultiple=false)]
public class ValidateInputAttribute : FilterAttribute, IAuthorizationFilter
{
    public ValidateInputAttribute(bool enableValidation)
    {
        this.EnableValidation = enableValidation;
    }

    public virtual void OnAuthorization(AuthorizationContext filterContext)
    {
        if (filterContext == null)
        {
            throw new ArgumentNullException("filterContext");
        }
        filterContext.Controller.ValidateRequest = this.EnableValidation;
    }

    public bool EnableValidation { get; private set; }
}
```

为了让读者对 `ValidateInputAttribute` 针对开启和关闭输入验证的作用有一个深刻的认识,我们来进行一个简单的实例演示。在一个 ASP.NET MVC 应用中定义了如下一个 `HomeController`, 包含在该 `Controller` 中的两个 `Action` 方法(`Action1` 和 `Action2`)具有两个字符串类型的参数 `foo` 和 `bar`, 其中 `Action1` 方法上应用了 `ValidateInputAttribute` 特性并将参数设置为 `False`。

```
public class HomeController : Controller
{
    [ValidateInput(false)]
    public void Action1(string foo, string bar)
    {
        Response.Write(string.Format("{0}: {1}<br/>", "foo",
            HttpUtility.HtmlEncode(foo)));
        Response.Write(string.Format("{0}: {1}<br/>", "bar",
            HttpUtility.HtmlEncode(bar)));
    }

    public void Action2(string foo, string bar)
    {
        Response.Write(string.Format("{0}: {1}<br/>", "foo",
            HttpUtility.HtmlEncode(foo)));
        Response.Write(string.Format("{0}: {1}<br/>", "bar",
            HttpUtility.HtmlEncode(bar)));
    }
}
```

直接运行该程序并在浏览器中通过输入相应的地址来访问这两个 `Action`, 我们以查询字

字符串的形式指定它们的两个参数。为了检验 ASP.NET MVC 对请求输入的验证,我们将表示参数 foo 的查询字符串的值设置为“<script></script>”。如图 7-11 所示, Action1 能够正常地被调用,而 Action2 在调用过程中抛出异常,并提示请求中包含危险的查询字符串。(S711)

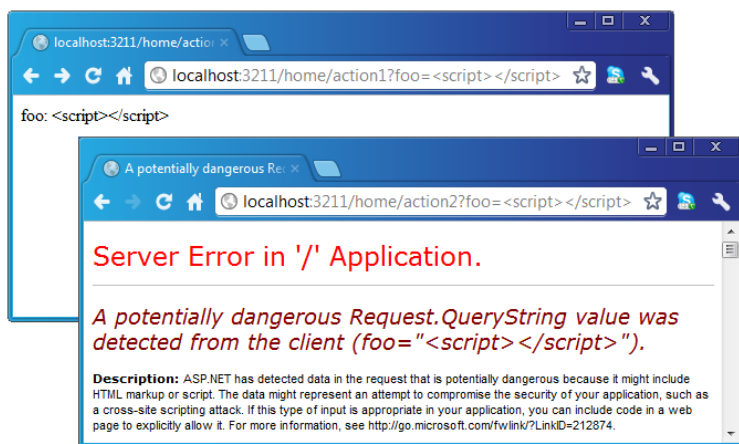


图 7-11 ValidateInputAttribute 对请求输入的验证

在第 5 章“Model 元数据的解析”中我们谈到过可以通过 AllowHtmlAttribute 特性来控制 ModelMetadata 的 RequestValidationEnabled 属性,从而忽略对目标属性成员内容的验证,使之可以包含 HTML 标签。这与 ValidateInputAttribute 的作用类似,不同的是 AllowHtmlAttribute 特性仅仅针对容器对象的某个数据(属性)成员,而 ValidateInputAttribute 则是针对整个请求。

ValidateAntiForgeryTokenAttribute

具有如下定义的 System.Web.Mvc.ValidateAntiForgeryTokenAttribute 用于解决一种叫做“跨站请求伪造(CSRF, Cross-Site Request Forgery)”的网络攻击,这是一种不同于 XSS (Cross Site Script) 的跨站脚本攻击,如果说 XSS 是利用了用户对网站的信任,而 CSRF 就是利用了站点对认证用户的信任。

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
    AllowMultiple=false, Inherited=true)]
public sealed class ValidateAntiForgeryTokenAttribute : FilterAttribute,
    IAuthorizationFilter
{
    public ValidateAntiForgeryTokenAttribute();
    public void OnAuthorization(AuthorizationContext filterContext);
    public string Salt { get; set; }
}
```

我们通过一个简单的例子来对 CSRF 的原理进行说明,假设我们通过 ASP.NET MVC 构建了一个博客应用,作为博主的用户可以发表博文,而一般用户(包括匿名用户)可以对博

文发表评论。除此之外，注册用户可以修改自己的 Email 地址，对应的操作定义在如下所示的 BlogController 的 Action 方法 UpdateEmailAddress 中。

```
public class BlogController: Controller
{
    [Authorize]
    [HttpPost]
    public void UpdateEmailAddress(string emailAddress)
    {
        //Email 地址修改操作
    }
    //其他成员
}
```

我们将 AuthorizeAttribute 特性应用在 UpdateEmailAddress 方法上，意味着只有认证的用户才能调用它来修改自己的 Email 地址。除此之外，HttpPostAttribute 特性也应用在该 Action 方法上，使我们只能以 POST 请求的方式调用它，这无形之中也增强了安全系数。但是这个方法提供的 Email 地址修改功能真的安全吗？它真能确保修改后的 Email 地址真的是登录用户提供的 Email 地址吗？

我们假设 BlogController 所在的 Web 应用部署的域名为 Foo，Action 方法 UpdateEmailAddress 对应的 URL 为 http://foo/blog/updateemailaddress。现在一个恶意攻击者创建如下一个简单的 HTML 页面，该页面具有一个指向上面这个地址的表单，并且该表单中具有一个名为 emailAddress 的元素提供属于攻击者自身的 Email 地址，由于注册了 window 的 onload 事件，该表单会在页面加载完成之后自动提交。

```
<html>
  <head>
    <script type="text/javascript">
      window.onload = function () {
        document.getElementById("updateEmail").submit();
      }
    </script>
  </head>
  <body>
    <form id="updateEmail" action="http://foo/blog/updateemailaddress"
      method="post">
      <input type="hidden" name="emailAddress" value="malicious@gmail.com" />
    </form>
  </body>
</html>
```

假设攻击者部署该页面的地址为 http://bar/maliciouspage.html，攻击者在某篇博文中添加一个包含如下 HTML（一张不能正常显示的图片）的评论，作为博主的我们在登录情况下打开这篇博文之后就会对这张图片的源地址发起请求，而定义在上面的这个表单被自动提交。

```
<imgsrc="http://bar/maliciouspage.html">...</img>
```

由于登录用户的安全令牌一般以 Cookie 形式存在，而该 Cookie 会存在于发送给针对 Action 方法 UpdateEmailAddress 的调用请求中（针对、<iframe>、<script>等标签在跨域访问中是否会发送非 Session Cookie 决于我们采用的浏览器），服务器会认为该请求来自被认证用户，所以最终造成了我们的 Email 地址被恶意修改而不自知。

这个例子充分说明了 CSRF 是一种比较隐蔽并且具有很大危害的网络攻击，促成攻击的原因在于服务器在执行目标操作的时候并没有验证请求的真正来源。对于 ASP.NET MVC 来说，如果我们在执行某个 Action 方法之前能够确认当前的请求来源的有效性，就能从根本上解决 CSRF 攻击。ValidateAntiForgeryTokenAttribute 结合 HtmlHelper 的 AntiForgeryToken 方法有效地解决了这个问题。

```
public class HtmlHelper
{
    //其他成员
    public MvcHtmlString AntiForgeryToken();
    public MvcHtmlString AntiForgeryToken(string salt);
    public MvcHtmlString AntiForgeryToken(string salt, string domain,
        string path);
}
```

如上面的代码片段所示，HtmlHelper 具有三个 AntiForgeryToken 方法（这里的方法是 HtmlHelper 的实例方法，不是扩展方法）。当我们在一个 View 中调用这些方法时，它们会为我们创建一个所谓“防伪令牌（Anti-Forgery Token）”的字符串，并以此生成一个类型为 hidden 的<input>元素。除此之外，该方法的调用还会根据这个防伪令牌设置一个具有 HttpOnly 标记的 Cookie。接下来就来详细讨论这个过程。

上述防伪令牌是通过一个 AntiForgeryData 对象生成的，如下面的代码片段所示，AntiForgeryData 是一个具有四个属性的内部类型，其核心是通过属性 Value 表示的“值”。属性 UserName 和 CreationDate 表示访问令牌授权的用户名和创建时间。字符串属性 Salt 是为了增强防伪令牌的安全系数，不同的 Salt 值会生成具有不同内容的防伪令牌，不同的防伪令牌在不同的地方被使用可以避免攻击者对一个防伪令牌的破解而使整个应用受到全面的攻击。ValidateAntiForgeryTokenAttribute 也具有一个同名的属性。

```
internal sealed class AntiForgeryData
{
    public string Value { get; set; }
    public string Salt { get; set; }
    public DateTime CreationDate { get; set; }
    public string Username { get; set; }
}
```

当 AntiForgeryToken 方法被调用时，它会先根据当前的请求的应用路径（对应 HttpRequest 的 ApplicationPath 属性）计算出表示防伪令牌 Cookie 的名称，该名称会在通过对应用路径进行 Base64 编码（编码之前需要进行一些特殊字符的替换工作）生成的字符串前添加“__RequestVerificationToken”前缀。

如果当前请求具有一个同名的 Cookie，则直接通过对 Cookie 的值进行反序列化得到一个 `AntiForgeryData` 对象。需要注意的是，这里针对 `AntiForgeryData` 进行序列化和反序列化并不是一个简单的在对象到字符串之间进行转换的过程，还包含采用 `MachineKey` 对 `AntiForgeryData` 的四个属性进行加密/解密的过程。

如果这样的 Cookie 不存在，`HtmlHelper` 会随机生成一个长度为 16 的字节数组，并对它进行 Base64 编码，然后创建一个 `AntiForgeryData` 对象并将这个编码后的字符串作为它的值（Value 属性值）。系统当前时间（UTC）作为该 `AntiForgeryData` 对象的创建时间，但是该 `AntiForgeryData` 对象的 `UserName` 和 `Salt` 属性为空。

接下来 `HtmlHelper` 会根据之前计算出来的 Cookie 名称创建一个 `HttpCookie` 对象，新创建出来的 `AntiForgeryData` 对象被序列化后生成的字符串作为该 `HttpCookie` 的值。如果我们在 `AntiForgeryToken` 方法调用时设置了表示域和路径的 `domain` 和 `path` 参数，它们将作为该 `HttpCookie` 对象的 `Domain` 和 `Path` 属性。`HtmlHelper` 最后将 `HttpCookie` 对象写入当前的 HTTP 响应（Set-Cookie）。

`AntiForgeryToken` 方法返回的是一个类型为 `hidden` 的 `<input>` 元素对应的 HTML，该 Hidden 元素的名称为 “__RequestVerificationToken”（即代码访问令牌 Cookie 名称的前缀）。为了生成该 Hidden 元素的值，`HtmlHelper` 会根据现有的 `AntiForgeryData` 对象（从当前请求获取的或者新创建的）创建一个新的 `AntiForgeryData` 对象，新旧两个对象具有相同的 `CreationDate` 和 `Value` 属性，而当前用户名和指定的 `Salt` 参数将会赋值给新 `AntiForgeryData` 对象的 `UserName` 和 `Salt` 属性（原 `AntiForgeryData` 不具有两个属性）。

```
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken("647B8734-EFCA-4F51-9D98-36502D13E4E7")
    ...
}
```

假如我们在一个 View 中通过如上所示的代码在一个表单中调用 `HtmlHelper` 的方法 `AntiForgeryToken`（将一个 GUID 作为 Salt），在最终生成的 HTML 中将会具有如下一个名为 “__RequestVerificationToken” 的 Hidden 元素。

```
<form action="..." method="post">
    <input name="__RequestVerificationToken" type="hidden"
value="yvLaFQ81JVgguKECyF/oQ+pc2/6q0MuLEaF73PvY7pvxaE68lO5qgXZWfhqIk721CBS
0SJZjvOjbc7o7GL3SQ3RxIW90no7FcxzR6ohHUYEKdxyfTBuAVjAuoil5miwoY8+6HNoSPbztY
hMVvtCsQDtvQfyWlGNa7qvlQSqYxQW7b6nAR2W00xNi4NgrFEqbMFrD+4CwwAg4PUWpvcQxYA=
" />
    ...
</form>
```

对于该 View 的首次访问或者对应的 Cookie 不存在，如下所示的一个名称为 “__RequestVerificationToken_L012Y0FwcDEX” 的 -Cookie（代表防伪令牌）将会出现在响应中。由于设置的 Cookie 具有 `HttpOnly` 标记，客户端（浏览器）是不能通过脚本获取到

Cookie 的值的。

```
HTTP/1.1 200 OK
Cache-Control: private
...
Set-Cookie:
__RequestVerificationToken_L012Y0FwcDEx=EYPOofprbB0og8vI+PzrlunY0Ye5BihYJg
oIYBqzvZDZ+hct5QUu+fj2hvFUVTTcFAZdjgCPzxwIGsoNdEyD8nSUbgapk8Xp3+ZD8cxguUrg
101AdFd4ZGWEYzz0IN5815saPJpuaChVR4QaMNbilNG4y7xiN2/UCrBF80LmPO4=; path=/;
HttpOnly
...
```

如果 ASP.NET MVC 确保其请求提交的表单具有一个名为“__RequestVerificationToken”的 Hidden 元素，并且该元素的值与对应的防伪令牌的 Cookie 值相匹配，就能够确保请求并不是由第三方恶意站点发送的，进而防止 CSRF 攻击。原因很简单，由于 Cookie 值是经过加密的，供给者可以得到整个 Cookie 的内容，但是不能解密获得具体的值（AntiForgeryData 的 Value 属性），所以不可能在提供的表单中也包含一个具有匹配值的 Hidden 元素。针对防伪令牌的验证就实现在 ValidateAntiForgeryTokenAttribute 的 OnAuthorization 方法中。

我们来具体介绍一下实现在 ValidateAntiForgeryTokenAttribute 中针对防伪令牌的验证逻辑。首先它根据当前请求的应用路径采用与生成防伪令牌 Cookie 相同的逻辑计算出 Cookie 名称。如果对应的 Cookie 不存在于当前请求中，则直接抛出 System.Web.Mvc.HttpAntiForgeryException 异常，否则获取 Cookie 值，并反序列化生成一个 AntiForgeryData 对象。

ValidateAntiForgeryTokenAttribute 从提交的表单中提取一个名为“__RequestVerificationToken”的输入元素，如果这样的元素不存在，同样抛出 HttpAntiForgeryException 异常，否则直接对具体的值进行反序列化生成一个 AntiForgeryData 对象。最后它在这两个 AntiForgeryData 的 Value 属性，以及将后者的 UserName 和 Salt 属性与当前用户名和自身的 Salt 属性进行比较，任何一个不匹配都会抛出 HttpAntiForgeryException 异常。

ChildActionOnlyAttribute

如果我们希望定义在 Controller 中的方法能以子 Action 的形式在某个 View 中被调用（这样的调用一般用于生成组成页面的某个部分的 HTML），可以在方法上应用具有如下定义的 System.Web.Mvc.ChildActionOnlyAttribute 特性。ChildActionOnlyAttribute 是一个 AuthorizationFilter，它在重写的 OnAuthorization 方法中对当前请求进行验证，对于非子 Action 调用它会直接抛出一个 InvalidOperationException 异常。

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
    AllowMultiple=false, Inherited=true)]
public sealed class ChildActionOnlyAttribute : FilterAttribute,
IAuthorizationFilter
{
    public void OnAuthorization(AuthorizationContext filterContext);
}
```

有的读者可能会问，AuthorizationFilter 如何区分当前的请求是基于子 Action 的调用，而不是一般的来自客户端的请求呢？其实很简单，当我们在调用 HtmlHelper 的扩展方法 Action/RenderAction 的时候会将当前的 ViewContext 作为“Parent ViewContext”保存到表示当前路由数据的 RouteData 的 DataTokens 属性中，对应的 Key 为“ParentActionViewContext”。如下面的代码片段所示，ControllerContext 的 IsChildAction 属性正是通过该路由信息来判断当前请求是否是针对子 Action 的调用。

```
public class ControllerContext
{
    //其他成员
    public virtual bool IsChildAction
    {
        get
        {
            RouteData routeData = this.RouteData;
            if (routeData == null)
            {
                return false;
            }
            return routeData.DataTokens.ContainsKey("ParentActionViewContext");
        }
    }
}
```

7.3.3 ActionFilter

在前面介绍 AsyncManager 的时候我们已经涉及到了两个 ActionFilter 的使用，它们是用于控制超时时限的 AsyncTimeoutAttribute 和 NoAsyncTimeoutAttribute。ActionFilter 允许我们在 Action 方法执行前后对调用进行拦截以执行一些额外的操作，ActionFilter 实现了具有如下定义的接口 System.Web.Mvc.IActionFilter。

```
public interface IActionFilter
{
    void OnActionExecuting(ActionExecutingContext filterContext);
    void OnActionExecuted(ActionExecutedContext filterContext);
}

public class ActionExecutingContext : ControllerContext
{
    public ActionExecutingContext();
    public ActionExecutingContext(ControllerContext controllerContext,
        ActionDescriptor actionDescriptor,
        IDictionary<string, object> actionParameters);

    public virtual ActionDescriptor ActionDescriptor { get; set; }
    public virtual IDictionary<string, object> ActionParameters { get; set; }
    public ActionResult Result { get; set; }
}

public class ActionExecutedContext : ControllerContext
{
}
```



```

public ActionExecutedContext();
public ActionExecutedContext(ControllerContext controllerContext,
    ActionDescriptor actionDescriptor, bool canceled, Exception exception);

public virtual ActionDescriptor ActionDescriptor { get; set; }
public virtual bool Canceled { get; set; }
public virtual Exception Exception { get; set; }
public bool ExceptionHandled { get; set; }
public ActionResult Result { get; set; }
}

```

如上面的代码片段所示, `IActionFilter` 接口具有 `OnActionExecuting` 和 `OnActionExecuted` 两个方法, 它们分别在目标 `Action` 方法执行前后被调用。这两个方法分别具有两个基于上下文的参数, 类型分别是 `System.Web.Mvc.ActionExecutingContext` 和 `System.Web.Mvc.ActionExecutedContext`, 它们均是 `ControllerContext` 的子类。

我们可以从 `ActionExecutingContext` 对象中获取到用于描述当前 `Action` 的 `ActionDescriptor` 和参数列表。`ActionFilter` 可以在 `OnActionExecuting` 方法中对 `ActionExecutingContext` 对象的 `Result` 属性进行赋值来对当前的请求实施响应。一旦 `ActionExecutingContext` 的 `Result` 属性被成功赋值, 将会终止后续 `ActionFilter` 和最终目标方法的执行。

`ActionExecutedContext` 具有额外的三个属性, 其中属性 `Exception` 表示执行 `Action` 方法过程中抛出的异常; 属性 `ExceptionHandled` 是一个表示是否对异常已经做出处理的标记; `Canceled` 属性表示没有完成整个 `ActionFilter` 链和目标 `Action` 方法的执行而中途被终止。

ActionFilter 的执行机制

当 `ActionInvoker` 在执行目标 `Action` 方法之前, 会根据 `Order` 和 `Scope` 属性对 `ActionFilter` 进行排序, 然后根据当前 `ControllerContext` 和 `ActionDescriptor` 创建一个 `ActionExecutingContext` 对象, 最后将其作为参数依次调用所有 `ActionFilter` 的 `OnActionExecuting` 方法。

`ActionFilter` 链中每一个 `ActionFilter` 的 `OnActionExecuting` 方法执行完毕之后, `ActionInvoker` 在执行目标 `Action` 方法之后会根据当前 `ControllerContext`、`ActionDescriptor` 以及 `Action` 方法执行过程中抛出的异常创建一个 `ActionExecutedContext` 对象。该 `ActionExecutedContext` 的 `Cancel` 属性被设置为 `False`。如果 `Action` 方法返回一个 `ActionResult` 对象, 该对象将会作为该 `ActionExecutedContext` 的 `Result` 属性。

接下来 `ActionInvoker` 按照相反顺序的调用 `ActionFilter` 链中每个 `ActionFilter` 的 `OnActionExecuted` 方法, 执行过程中的 `ActionFilter` 可以修改 `ActionExecutedContext` 的 `Result` 属性。当整个 `ActionFilter` 链执行结束之后, `ActionExecutedContext` 的 `Result` 属性返回的 `ActionResult` 将会用于针对请求的响应。图 7-12 基本上反映了整个 `ActionFilter` 链的执行过程。

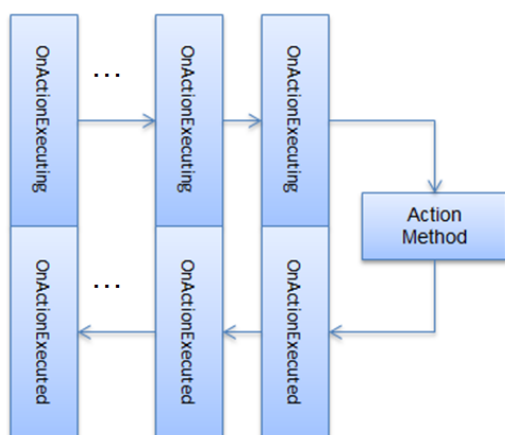


图 7-12 连同目标 Action 在内的整个 ActionFilter 链的执行

ActionFilter 对 ActionResult 的设置

上面我们已经提到过，在调用 ActionFilter 的 OnActionExecuting 方法的过程中，一旦某个 ActionFilter 为 ActionExecutingContext 的 Result 属性设置了一个 ActionResult 对象，后续 ActionFilter 和目标 Action 将不会被执行。此时 ActionInvoker 会创建一个 ActionExecutedContext 对象，设置的 ActionResult 直接作为其 Result 属性，而 Cancel 属性被设置为 True。现在要考虑的问题是之前的 ActionFilter 的 OnActionExecuted 是否还会执行呢？

我们不妨使用一个简单的演示实例来寻求问题的答案。在一个 ASP.NET MVC 应用中定义了如下三个 ActionFilter（FooAttribute、BarAttribute 和 BazAttribute），它们都继承自自定义的 FilterBaseAttribute。在 FilterBaseAttribute 中实现的 OnActionExecuting 和 OnActionExecuted 方法中，我们将 ActionFilter 自身的类型和执行方法名写入当前 HttpResponse 并最终呈现在浏览器中。BarAttribute 重写了 OnActionExecuting 方法，在调用基类同名方法之后为 ActionExecutingContext 的 Result 设置了一个 EmptyResult 对象。

```

public abstract class FilterBaseAttribute : FilterAttribute, IActionFilter
{
    public virtual void OnActionExecuted(ActionExecutedContext filterContext)
    {
        filterContext.HttpContext.Response.Write(
            string.Format("{0}.OnActionExecuted()<br/>", this.GetType().Name));
    }

    public virtual void OnActionExecuting(ActionExecutingContext filterContext)
    {
        filterContext.HttpContext.Response.Write(
            string.Format("{0}.OnActionExecuting()<br/>", this.GetType().Name));
    }
}

```

```

public class FooAttribute : FilterBaseAttribute
{
}
public class BarAttribute : FilterBaseAttribute
{
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        base.OnActionExecuting(filterContext);
        filterContext.Result = new EmptyResult();
    }
}
public class BazAttribute : FilterBaseAttribute
{
}

```

然后我们定义了如下一个 `HomeController`，上面三个 `ActionFilter` 特性同时被应用到了 `Action` 方法 `Index` 上。我们对三个 `ActionFilter` 特性的 `Order` 属性作了相应地设置使它们可以按照我们希望的顺序（`FooAttribute` => `BarAttribute` => `BazAttribute`）执行。

```

public class HomeController : Controller
{
    [Foo(Order = 1)]
    [Bar(Order = 2)]
    [Baz(Order = 3)]
    public void Index()
    {
        Response.Write("Index...<br>");
    }
}

```

运行该程序后会在浏览器中呈现出如图 7-13 所示的输出结果。根据这个输出结果可以看出应用到同一个 `Action` 方法上的三个 `ActionFilter` 按照 `Order` 属性构建的 `ActionFilter` 链。在执行 `OnActionExecuting` 方法的过程中，处于中间位置的 `BarAttribute` 将 `ActionExecutingContext` 的 `Result` 属性进行了相应设置后，位于它之前的 `FooAttribute` 的 `OnActionExecuted` 方法依然会执行。（S712）

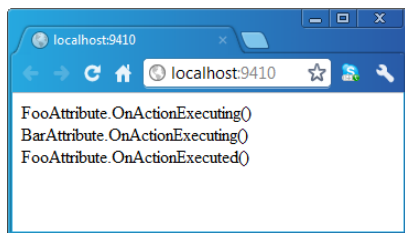


图 7-13 ActionFilter 链执行流程

这个简单的实例揭示了应用到同一个 `Action` 方法上的 `ActionFilter` 链的执行机制：如果某个 `ActionFilter` 在执行 `OnActionExecuting` 方法过程中对 `ActionExecutingContext` 的 `Result` 属性进行了设置，后续的 `ActionFilter` 和目标 `Action` 方法将不会再执行。此时 `ActionExecutedContext` 对象被创建，`ActionExecutingContext` 的 `Result` 属性表示的 `ActionResult` 对象将会作为它的 `Result` 属性。

接下来 ASP.NET MVC 会以前一个 `ActionFilter` 作为起点逆向执行 `ActionFilter` 链的 `OnActionExecuted` 方法，而作为参数的自然就是这个 `ActionExecutedContext` 对象。图 7-14 基本上揭示了整个 `ActionFilter` 链执行的流程。

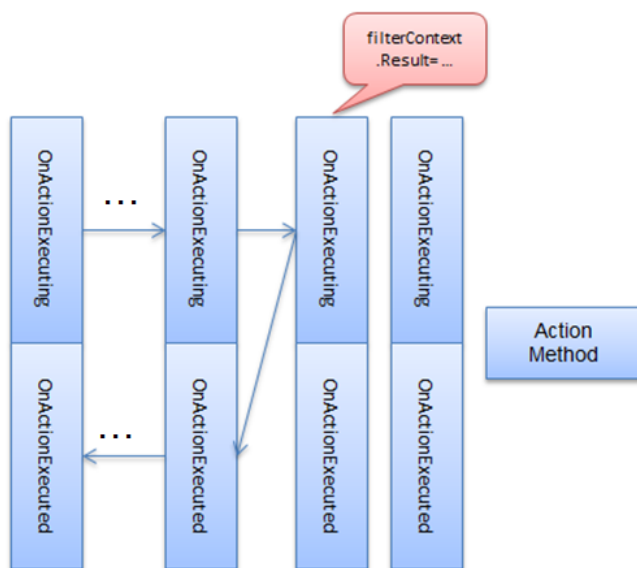


图 7-14 在 Action 方法执行前对 Result 的设置对整个 `ActionFilter` 链执行的影响

如果在逆向执行 `ActionFilter` 链的 `OnActionExecuted` 方法过程中某个 `ActionFilter` 对 `ActionExecutedContext` 的 `Result` 属性作了相应的设置，后续的 `ActionFilter` 依然按照相应的次序正常执行。

ActionFilter 中的异常处理

前面我们讨论了 `ActionFilter` 链在执行过程中某个 `ActionFilter` 分别在执行 `OnActionExecuting` 和 `OnActionExecuted` 方法时分别对 `ActionExecutingContext` 和 `ActionExecutedContext` 的 `Result` 进行相应设置具有怎样的影响，现在我们讨论如果某个 `ActionFilter` 在执行 `OnActionExecuting` 和 `OnActionExecuted` 方法过程中抛出异常后，后续的工作又将如何进行。

如果 `ActionFilter` 链的第一个 `ActionFilter` 在执行 `OnActionExecuting` 或者 `OnActionExecuted` 方法的过程中出现异常，那么这个异常会被直接抛出。如果抛出异常的并不是第一个 `ActionFilter`，抛出异常会被捕捉。`ActionInvoker` 会创建一个 `ActionExecutedContext` 对象，抛出的异常直接作为它的 `Exception` 属性。随后这个 `ActionExecutedContext` 对象被作为参数调用前一个 `ActionFilter` 的 `OnActionExecuted` 方法，如果在执行过程中这个 `ActionFilter` 将 `ActionExecutedContext` 的 `ExceptionHandled` 属性设置为 `True`，表明抛出的异常已经经过处理，

那么 `ActionInvoker` 会按照正常的方式逆向调用后续 `ActionFilter` 链(整个 `ActionFilter` 链中位于当前 `ActionFilter` 之前的部分)。

如果 `ActionFilter` 在执行 `OnActionExecuted` 之后 `ActionExecutedContext` 的 `ExceptionHandled` 属性依然是 `False`，之前捕获的异常会再次抛出来。`ActionInvoker` 又会捕获这个异常创建一个 `ActionExecutedContext` 对象作为参数调用前面的 `ActionFilter` 的 `OnActionExecuted` 方法。如果异常是在非链头的 `ActionFilter` 的 `OnActionExecuted` 方法中抛出，处理流程与此类似。

我们不妨举例说明 `Action` 链在执行过程中对异常的处理。假设具有如图 7-15 所示的包含四个节点的 `ActionFilter` 链，其 `Filter1`、`Filter2` 和 `Filter3` 的 `OnActionExecuting` 方法先后正常执行，但是 `Filter4` 在执行 `OnActionExecuting` 方法的时候抛出一个异常。这个异常会被 `ActionInvoker` 捕获并据此创建一个 `ActionExecutedContext` 对象，`ActionInvoker` 随后会将它作为参数调用 `Filter3` 的 `OnActionExecuted` 方法（步骤 1）。

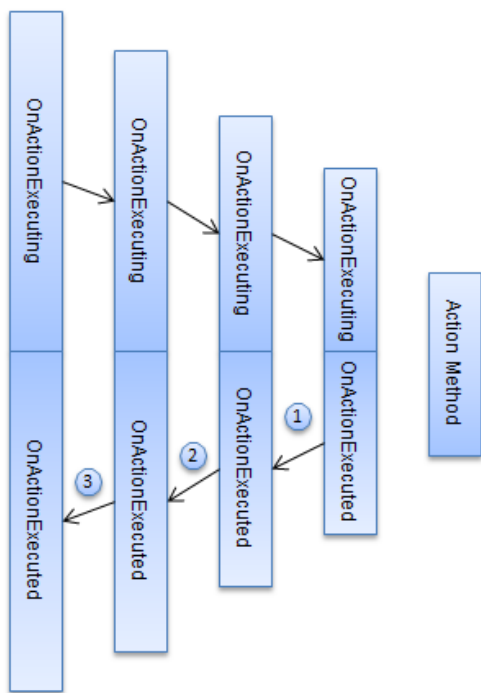


图 7-15 `ActionFilter` 链执行过程中对异常的处理

如果 `Filter3` 在执行 `OnActionExecuted` 方法后 `ActionExecutedContext` 的 `ExceptionHandled` 属性为 `False`，之前捕获的异常会再次抛出来。`ActionInvoker` 又会按照相同的方式根据抛出的异常创建一个 `ActionExecutedContext` 对象，并作为参数调用 `Filter2` 的 `OnActionExecuted` 方法（步骤 2）。

如果 Filter2 在执行 `OnActionExecuted` 方法的过程中将 `ActionExecutedContext` 的 `ExceptionHandled` 属性设置为 `True`, 表明抛出的异常已经经过合理的处理, `ActionInvoker` 会按照正常的方式调用 Filter1 的 `OnActionExecuted` 方法 (步骤 3)。如果 Filter1 在执行 `OnActionExecuted` 过程中不出现异常, 最终是不会有异常抛出的。

7.3.4 ExceptionFilter

`ExceptionFilter` 是一个用于进行异常处理的筛选器。`ExceptionFilter` 不仅仅用于处理执行整个 `ActionFilter` 链 (包括目标 `Action` 方法的执行) 最终抛出的异常, 还用于处理执行整个 `ResultFilter` 链 (包括对 `Action` 方法返回的 `ResultFilter` 的执行) 最终抛出的异常。`ExceptionFilter` 实现了具有如下定义的接口 `System.Web.Mvc.IExceptionFilter`。

```
public interface IExceptionFilter
{
    void OnException(ExceptionContext filterContext);
}

public class ExceptionContext : ControllerContext
{
    public ExceptionContext(ControllerContext controllerContext,
        Exception exception);

    public virtual Exception Exception { get; set; }
    public bool ExceptionHandled { get; set; }
    public ActionResult Result { get; set; }
}
```

如上面的代码片段所示, `IExceptionFilter` 具有唯一的方法 `OnException` 用于进行异常处理, 该方法的参数是一个类型为 `System.Web.Mvc.ExceptionContext` 的上下文对象。`ExceptionContext` 同样是 `ControllerContext` 的子类, 它的 `Exception` 表示抛出的异常, 而 `ExceptionHandled` 属性表示是否已经完成了对异常的处理。如果需要对请求作出响应, 需要为 `ExceptionContext` 的 `Result` 属性设置一个 `ActionResult` 对象。我们可以设置一个 `ViewResult` 显示一个错误页面; 对于 Ajax 请求, 可以返回一个包含异常信息的 `JsonResult` 对象。

最终应用到某个 `Action` 方法上的多个 `ExceptionFilter` 根据 `Order` 和 `Scope` 属性排列成一个 `ExceptionFilter` 链, 对于 `ExceptionFilter` 的执行, 有如下三点需要着重强调。

- `ExceptionFilter` 链是反向执行的。对于根据 `Order` 和 `Scope` 属性排好序的 `ExceptionFilter` 链, 排在后面的具有更高的执行优先级。
- 将 `ExceptionContext` 的 `ExceptionHandled` 设置为 `True` 并不能阻止后续 `ExceptionFilter` 的执行。
- 如果 `ExceptionFilter` 在执行 `OnException` 过程中出现异常, 整个 `ExceptionFilter` 链的执行将立即终止, 并且该异常会被直接抛出来。

HandleErrorAttribute

ASP.NET MVC 提供了一个 `System.Web.Mvc.HandleErrorAttribute` 使我们可以针对具体的异常类型来呈现对应的错误页面。如下面的代码片段所示, `HandleErrorAttribute` 具有一个表示被处理异常类型的 `ExceptionType` 属性, 只有在被处理异常与该类型匹配的情况下通过 `HandleErrorAttribute` 指定的作为错误页面的 `View` 才会呈现出来。该属性的默认值为 `System.Exception` 类型, 意味着默认情况下 `HandleErrorAttribute` 可以处理所有类型的异常。

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, Inherited=true,
    AllowMultiple=true)]
public class HandleErrorAttribute : FilterAttribute, IExceptionFilter
{
    public virtual void OnException(ExceptionContext filterContext);

    public Type           ExceptionType { get; set; }
    public string         Master { get; set; }
    public string         View { get; set; }
    public override object TypeId { get; }
}
```

`HandleErrorAttribute` 对异常的处理策略就是基于异常类型的错误页面的呈现, 它的属性 `View` 和 `Master` 表示作为错误页面的 `View` 名称和对应的布局文件名, 默认值分别为 “Error” 和空字符串。 `HandleErrorAttribute` 在进行 `View` 呈现的时候会根据当前 `Controller` 和 `Action` 的名称以及抛出的异常创建一个具有如下定义的 `System.Web.Mvc.HandleErrorInfo` 对象作为其 `Model`, 所以我们定义一个 `Model` 类型为 `HandleErrorInfo` 的 `View` 来显示相应的错误和上下文信息。

```
public class HandleErrorInfo
{
    public HandleErrorInfo(Exception exception, string controllerName,
        string actionName);

    public string     ActionName { get; private set; }
    public string     ControllerName { get; private set; }
    public Exception  Exception { get; private set; }
}
```

由于应用在 `HandleErrorAttribute` 上面的 `AttributeUsageAttribute` 的 `AllowMultiple` 属性被设置为 `True`, 所以 ASP.NET MVC 允许我们按照如下的方式在同一个 `Action` 方法或者 `Controller` 类型上应用多个 `HandleErrorAttribute` 特性, 以实现对不同异常类型的针对性处理 (根据抛出异常类型的不同, 显示不同的错误页面)。

```
public class FooController : Controller
{
    [HandleError(ExceptionType = typeof(Exception1), View="ErrorView1",
        Order=3)]
    [HandleError(ExceptionType = typeof(Exception2), View = "ErrorView2",
        Order=2)]
    [HandleError(Order = 1)]
    public ActionResult Bar()
    {
```

```

        //省略操作
    }
}

```

针对 `ExceptionHandler` 链反向执行的特性，我们需要通过设置 `Order` 属性让针对具体异常类型的 `HandleErrorAttribute` 优先执行。以上面的代码片段为例，如果 `Exception1` 继承自 `Exception2`，我们必须让针对 `Exception1` 的 `HandleErrorAttribute` 先于针对 `Exception2` 的 `HandleErrorAttribute` 执行，而针对 `System.Exception`（默认值）的 `HandleErrorAttribute` 最后执行。

如果反过来，针对 `Exception1` 和 `Exception2` 的 `HandleAttribute` 特性将变得毫无意义。因为 `HandleAttribute` 的异常处理工作只有在当前 `ExceptionContext` 的 `ExceptionHandled` 属性为 `False` 的时候才会进行，而它在进行异常处理的时候总会将该属性设置为 `True`。

当我们通过 Visual Studio 的 ASP.NET MVC 项目模板创建一个 Web 应用的时候，针对 `HandleErrorAttribute` 的注册代码默认出现在生成的 `FilterConfig`（该类型所在文件在 `App_Start` 目录下）类型的 `RegisterGlobalFilters` 方法中，具体的注册代码如下所示。

```

public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
    }
}

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    }
}

```

关于 `HandleErrorAttribute` 还有一点需要着重强调的是，只有在当前 `HttpContext` 用于表示是否可以对错误进行定制的 `IsCustomErrorEnabled` 属性为 `True` 的情况下，`HandleErrorAttribute` 才会真正被用于处理抛出的异常，可以通过如下的配置来控制是否允许对错误的定制。

```

<configuration>
  <system.web>
    <customErrors mode="On|Off|RemoteOnly"/>
  </system.web>
</configuration>

```

7.3.5 实例演示：集成 EntLib 实现自动化异常处理 (S713, S714, S715)

个人觉得异常处理对于程序员来说是最为熟悉的同时也可能是最难掌握的。说它熟悉，

是因为异常处理的编程模式仅仅是 try/catch/finally 而已；说它难以掌握，则是因为很多开发人员往往说不清楚 try/catch/finally 应该置于何处？什么情况下需要对异常进行日志记录？什么情况下需要对异常进行封装？什么情况下需要对异常进行替换？对于捕获的异常，在什么情况下需要将其再次抛出？什么情况下又不需要再次抛出。

合理的异常处理应该是场景驱动的，在不同的场景下采用的异常处理策略往往是不同的。异常处理的策略最好是可配置的，因为应用程序出现怎样的异常往往是不可预测的，现有异常策略的不足往往需要在真正出现某种异常的时候才会体现出来，所以我们需要一种动态可配置的异常处理策略维护方式。目前有一些开源的异常处理框架提供了这种可配置的、场景驱动的异常处理方式，微软企业库（EntLib）的 Exception Handling Application Block（以下简称 EHAB）就是一个不错的选择。

EntLib 中的 EHAB

微软企业库异常处理应用块（EHAB）采用基于“策略”的异常处理机制，异常处理策略通过配置定义。EHAB 中的异常处理策略大致可以通过下面的公式表示。

异常处理策略（Exception Handling Policy）= 异常类型（Exception Type）+ 异常处理器（Exception Handler）+ 异常后续处理方式（Post Handling Action）

EHAB 中的异常处理策略表达的意思是，当出现某种类型的异常时，应该采用怎样的方式处理，以及在处理之后是否抛出原始异常或者处理后异常。EHAB 的异常处理机制是基于“类型”的，而异常处理逻辑则实现在一个个异常处理器中（Exception Handler）。异常后续处理方式主要分为三种情形：抛出原始异常、抛出处理后的异常和不做任何操作。这三种处理方式定义在 Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.PostHandling Action 枚举中。

```
public enum PostHandlingAction
{
    None,
    NotifyRethrow,
    ThrowNewException
}
```

EHAB 异常处理策略可以通过配置的方式定义。下面的配置中演示了针对 SqlException 的处理，我们通过 LoggingExceptionHandler 和 ReplaceHandler 这两个异常处理器先后对抛出的 SqlException 异常进行处理，前者对进行抛出的异常进行日志记录，后者将其替换成自定义的 DbException。当抛出的异常先后被这两个异常处理器处理之后，替换后的 DbException 会被抛出来（postHandlingAction="ThrowNewException"）。

```
<configuration>
...
<exceptionHandling>
  <exceptionPolicies>
    <add name="data access policy">
```

```

<exceptionTypes>
  <add type="System.Data.SqlClient.SqlException, System.Data"
    postHandlingAction="ThrowNewException" name="SqlException">
    <exceptionHandlers>
      <add name="Logging Handler"
        type="Microsoft.Practices.EnterpriseLibrary
          .ExceptionHandling.Logging.LoggingExceptionHandler"../>
      <add name="Replace Handler"
        type="Microsoft.Practices.EnterpriseLibrary
          .ExceptionHandling.ReplaceHandler"
        exceptionMessage="Encounter data access error."
        replaceExceptionType=
          "Artech.DataAccess.DbException, Artech.DataAccess"/>
    </exceptionHandlers>
  </add>
</exceptionTypes>
</add>
</exceptionPolicies>
</exceptionHandling>
</configuration>

```

由于异常处理策略完全定义在配置中，在编程的时候，我们仅仅需要按照下面的方式指定相应的策略名称即可（data access policy）。关于 EHAB，由于篇幅的问题，只能点到为止，有兴趣的读者可以参阅微软企业库开发文档。

```

try
{
    return this.MembershipProxy.ValidateUser(username, password);
}
catch (Exception ex)
{
    if (ExceptionPolicy.HandleException(ex, "data access policy"))
    {
        throw;
    }
}

```

“自动化”异常处理

在正式介绍如何通过扩展 EntLib 的 EHAB 进行集成以实现自动化异常处理之前，我们不妨先来体验一下异常处理到底怎么个“自动化”法。以用户登录场景为例，我们在一个 ASP.NET MVC 应用中定义了如下一个简单的数据类型 LoginInfo，两个属性 UserName 和 Password 表示登录输入的用户名和密码。

```

public class LoginInfo
{
    [DisplayName("用户名")]
    [Required(ErrorMessage="请输入{0}")]
    public string UserName { get; set; }

    [DisplayName("密码")]
    [Required(ErrorMessage = "请输入{0}")]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}

```

然后定义了如下一个 HomeController。基于 HTTP-GET 的 Action 方法 Index 将会呈现一个用户登录 View，而真正的用户验证逻辑定义在另一个应用了 HttpPostAttribute 特性的 Index 方法中。具体的认证逻辑很简单，如果用户名不为“Foo”，抛出 InvalidUserNameException 异常；如果密码不是“password”，则抛出 InvalidPasswordException 异常。InvalidUserNameException 和 InvalidPasswordException 是我们自定义的两种异常类型。

```
[ExceptionPolicy("defaultPolicy")]
public class HomeController : ExtendedController
{
    public ActionResult Index()
    {
        return View(new LoginInfo());
    }

    [HttpPost]
    [HandleErrorAction("OnIndexError")]
    public ActionResult Index(LoginInfo loginInfo)
    {
        if (string.Compare(loginInfo.UserName, "foo", true) != 0)
        {
            throw new InvalidUserNameException();
        }

        if (loginInfo.Password != "password")
        {
            throw new InvalidPasswordException();
        }
        return View(loginInfo);
    }

    [HttpPost]
    public ActionResult OnIndexError(LoginInfo loginInfo)
    {
        return View(loginInfo);
    }
}
```

上面定义的 HomeController 具有三点与自动化异常处理相关的地方。

- HomeController 继承自自定义的基类 ExtendedController，后者完成了对异常的自动化处理。
- HomeController 类型上应用了自定义的 ExceptionPolicyAttribute 特性用于指定默认采用的异常处理策略名称（“defaultPolicy”）。
- 应用了 HttpPostAttribute 特性的 Index 方法上标注了一个 HandleErrorActionAttribute 特性用于指定一个 Handle-Error-Action 名称。在目标 Action 执行过程中抛出的异常被 EHAB 处理后，指定的 Action 会被执行以实现请求的响应。对于我们的例子来说，从 Index 方法抛出的异常被处理后会调用 OnIndexError 方法并利用返回的 ActionResult 来响应当前请求。

下面是代表登录页面的 View 的定义，这是一个 Model 类型为 LoginInfo 的强类型 View。

在该 View 中，作为 Model 的 LoginInfo 对象以编辑模式呈现在一个表单中，表单中提供了一个“登录”按钮提交表单。除此之外，这个 View 中还具有一个 ValidationSummary。

```
@model LoginInfo
<html>
  <head>
    <title>用户登录</title>
    <style type="text/css">
      .validation-summary-errors{color:Red}
    </style>
  </head>
  <body>
    @using (Html.BeginForm())
    {
      @Html.ValidationSummary(true)
      @Html.EditorForModel()
      <input type="submit" value="登录" />
    }
  </body>
</html>
```

通过 HomeController 的定义知道，两种不同类型的异常（InvalidUserNameException 和 InvalidPasswordException）分别在输入无效用户名和密码时被抛出来，而我们需要处理的就是这两种类型的异常。针对这两种类型异常的处理策略定义在如下的配置中，策略名称就是通过应用在 HomeController 上的 ExceptionPolicyAttribute 特性指定的“defaultPolicy”。

```
<configuration>
  <configSections>
    <section name="exceptionHandling"
      type="Microsoft.Practices.EnterpriseLibrary.ExceptionHandling
        .Configuration.ExceptionHandlingSettings,
        Microsoft.Practices.EnterpriseLibrary.ExceptionHandling" />
  </configSections>
  <exceptionHandling>
    <exceptionPolicies>
      <add name="defaultPolicy">
        <exceptionTypes>
          <add type="MvcApp.InvalidUserNameException, MvcApp"
            postHandlingAction="ThrowNewException"
            name="InvalidUserNameException">
            <exceptionHandlers>
              <add name="ErrorMessageHandler"
                type="MvcApp.ErrorMessageHandler, MvcApp"
                errorMessage="用户名不存在"/>
            </exceptionHandlers>
          </add>

          <add type="MvcApp.InvalidPasswordException, MvcApp"
            postHandlingAction="ThrowNewException"
            name="InvalidPasswordException">
            <exceptionHandlers>
              <add name="ErrorMessageHandler"
                type="MvcApp.ErrorMessageHandler, MvcApp"
                errorMessage="密码与用户名不匹配"/>
            </exceptionHandlers>
          </add>
        </exceptionTypes>
      </add>
    </exceptionPolicies>
  </exceptionHandling>
</configuration>
```

```

        </add>
    </exceptionTypes>
</add>
</exceptionPolicies>
</exceptionHandling>
...
</configuration>

```

如上面的配置片段所示，我们使用一个自定义的 `ErrorMessageHandler` 来处理抛出来的 `InvalidUserNameException` 和 `InvalidPasswordException` 异常，而 `ErrorMessageHandler` 仅仅是指定一个友好的错误消息而已。

运行该程序后一个登录页面会被呈现出来，当我们输入错误的用户名和密码的时候，相应的错误消息（在配置中通过 `ErrorMessageHandler` 设置的错误消息）会以如图 7-16 所示的效果显示在当前 View 的 `ValidationSummary` 中。其实整个 View 是通过执行 `Action` 方法 `OnIndexError` 返回的 `ViewResult` 呈现出来的。（S713）

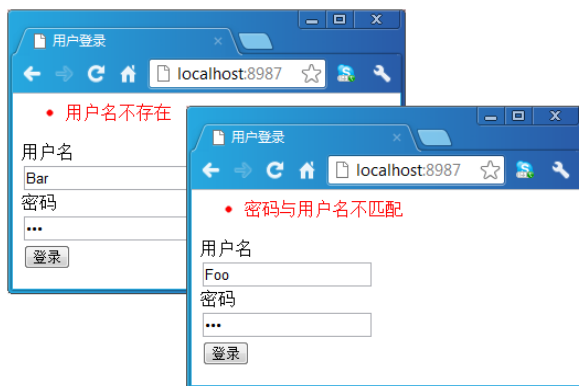


图 7-16 通过执行 `Handle-Error-Action` 呈现异常处理结果

除了通过执行对应的 `Handle-Error-Action` 来呈现异常处理后的最终结果之外，还支持错误页面的错误呈现方法。简单起见，我们直接将作为错误页面的 View 名称设置为“Error”。为了演示基于错误页面的呈现方式，我们按照如下的方式在“`\Views\Shared\`”目录下定义了如下一个名为 `Error` 的 View。

```

@model ExtendedHandleErrorInfo
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Error</title>
    <style type="text/css">
        h3 {color:Red;}
    </style>

```

```

</head>
<body>
  <h3>
    @Html.DisplayFor(m=>m.ErrorMessage)
  </h3>
  <ul>
    <li>Controller: @Html.DisplayFor(m => m.ControllerName)</li>
    <li>Action: @Html.DisplayFor(m => m.ActionName)</li>
    <li>Exception:
      <ul>
        <li>Message: @Html.DisplayFor(m => m.Exception.Message)</li>
        <li>Type: @Model.Exception.GetType().FullName</li>
        <li>StackTrace: @Html.DisplayFor(
          m => m.Exception.StackTrace)</li>
      </ul>
    </li>
  </ul>
</body>
</html>

```

上面这个作为错误页面的 View 使用具有如下定义的 ExtendedHandleErrorInfo 类型作为其 Model。ExtendedHandleErrorInfo 继承自 HandleErrorInfo，它只额外定义了一个表示错误消息的 ErrorMessage 属性。在上面的这个 View 中，我们将错误消息、异常类型、StackTrace、当前 Controller 和 Action 的名称呈现出来。

```

public class ExtendedHandleErrorInfo : HandleErrorInfo
{
    public string ErrorMessage { get; private set; }
    public ExtendedHandleErrorInfo(Exception exception,
        string controllerName, string actionName, string errorMessage)
        : base(exception, controllerName, actionName)
    {
        this.ErrorMessage = errorMessage;
    }
}

```

如果我们采用错误页面的方式来响应请求，需要按照如下的方式将应用在 Action 方法 Index 上的 HandleErrorActionAttribute 特性注释掉。

```

[ExceptionHandler("defaultPolicy")]
public class HomeController : ExtendedController
{
    //其他成员
    [HttpPost]
    //[[HandleErrorAction("OnIndexError")]]
    public ActionResult Index(LoginInfo loginInfo)
    {
        //省略实现
    }
}

```

再次运行该程序并在用户登录页面上分别输入错误的用户名和密码后，默认的错误页面（Error.cshtml）将会以如图 7-17 所示的效果把处理后的异常信息呈现出来。（S714）

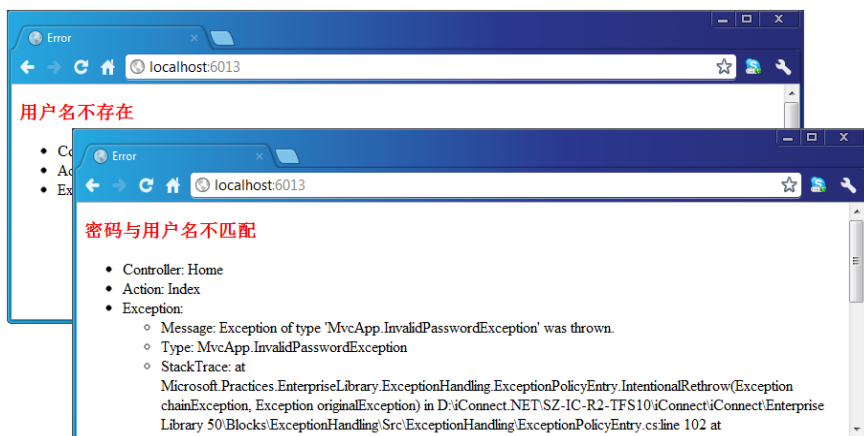


图 7-17 通过执行错误 View 呈现异常处理结果

用于实施认证的 Action 方法 Index 也可以通过 Ajax 请求的方式来调用。对于 Ajax 请求来说，我们会将通过 EntLib 处理后的异常封装成如下一个类型为 `ExceptionDetail` 的对象，它具有与 `Exception` 对应的属性设置，最终根据这个 `ExceptionDetail` 对象创建一个 `JsonResult` 来响应当前的请求。

```
public class ExceptionDetail
{
    public ExceptionDetail(Exception exception, string errorMessage=null)
    {
        this.HelpLink          = exception.HelpLink;
        this.Message            = string.IsNullOrEmpty(errorMessage)
                                ? exception.Message : errorMessage;
        this.StackTrace          = exception.StackTrace;
        this.ExceptionType       = exception.GetType().ToString();
        if (exception.InnerException != null)
        {
            this.InnerException = new ExceptionDetail(exception.InnerException);
        }
    }

    public string                HelpLink { get; set; }
    public ExceptionDetail       InnerException { get; set; }
    public string                Message { get; set; }
    public string                StackTrace { get; set; }
    public string                ExceptionType { get; set; }
}
```

当客户端接收到响应的 Json 对象后，可以通过检测其是否具有一个 `ExceptionType` 属性（对于一个 `ExceptionDetail` 对象来说，该属性不可能为 `Null`）来判断是否发生异常。作为演示，我们对 Action 方法 Index 对应的 View 进行了如下改动。

```
@model LoginInfo
<html>
<head>
    <title>用户登录</title>
```

```

<script type="text/javascript"
    src="@Url.Content("~/Scripts/jquery-1.7.1.js")"></script>
<script type="text/javascript"
    src="@Url.Content("~/Scripts/jquery.unobtrusive-ajax.js")"></script>
<script type="text/javascript">
    function login(data) {
        if (data.ExceptionType) {
            alert(data.Message);
        }
        else {
            alert("认证成功");
        }
    }
</script>
</head>
<body>
    @{
        AjaxOptions options = new AjaxOptions{OnSuccess = "login"};
    }
    @using (Ajax.BeginForm(options))
    {
        @Html.EditorForModel()
        <input type="submit" value="登录" />
    }
</body>
</html>

```

如上面的代码片段所示，通过调用 `AjaxHelper` 的 `BeginForm` 生成了一个以 Ajax 形式提交的表单。表单成功提交（服务端因对抛出的异常进行处理而响应一个封装了异常信息的 `Json` 对象，对于提交表单的 Ajax 请求来说依然属于成功提交）后会调用我们定义的回调函数 `login`。在该 JavaScript 函数中，我们通过得到的对象是否具有一个 `ExceptionType` 属性来判断服务端是否抛出异常。如果抛出异常，则通过调用 `alert` 方法将错误信息显示出来，否则显示“认证成功”。

再次运行我们的程序并分别输入不合法的用户名和密码，相应的错误消息会以对话框的形式显示出来，具体的显示效果如图 7-18 所示。（S715）

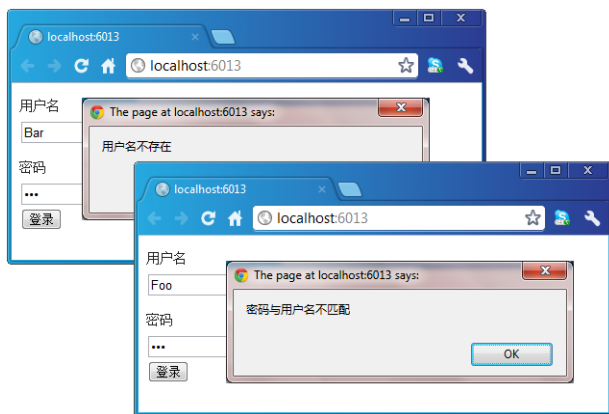


图 7-18 针对 Ajax 请求的错误消息的呈现

ExtendedController

通过上面的实例演示可以看出我们的扩展能够利用 EntLib 的 EHAB 根据指定的异常处理策略对抛出的异常进行处理。而对于处理后的结果，它会按照如下的策略来响应请求。

- 对于 Ajax 请求，直接创建一个用于封装被处理后异常的数据对象，并据此创建一个 JsonResult 对象来响应当前请求。
- 对于非 Ajax 请求，如果当前 Action 方法上通过应用的 HandleErrorActionAttribute 特性设置了匹配的 Action 方法，这个 Action 方法会自动被执行并采用返回的 ActionResult 对象响应当前请求。
- 如果 HandleErrorActionAttribute 特性不曾应用在当前 Action 方法上，或者通过该特性指定的 Action 根本不存在，则将作为错误页面的 View 呈现出来作为对请求的响应。

所有的这些都是通过一个自定义的 ExceptionFilter 来实现的。不过我们并没有定义任何的 ExceptionFilter 特性，而是将异常处理实现在一个自定义的 ExtendedController 基类中，对异常的自动处理实现在重写的 OnException 方法中。在介绍定义在该方法中具体的异常处理逻辑之前，我们先来看看定义在 ExtendedController 中的其他辅助成员。

```
public class ExtendedController: Controller
{
    private static Dictionary<Type, ControllerDescriptor> controllerDescriptors
        = new Dictionary<Type, ControllerDescriptor>();
    private static object syncHelper = new object();

    protected override void OnException(ExceptionContext filterContext)
    {
        //省略成员
    }

    //描述当前 Controller 的 ControllerDescriptor
    public ControllerDescriptor Descriptor
    {
        get
        {
            ControllerDescriptor descriptor;
            if (controllerDescriptors.TryGetValue(this.GetType(), out descriptor))
            {
                return descriptor;
            }
            lock (syncHelper)
            {
                if (controllerDescriptors.TryGetValue(this.GetType(),
                    out descriptor))
                {
                    return descriptor;
                }
                else
                {
                    descriptor =
                        new ReflectedControllerDescriptor(this.GetType());
                    controllerDescriptors.Add(this.GetType(), descriptor);
                    return descriptor;
                }
            }
        }
    }
}
```

```

    }
}

//获取异常处理策略名称
public string GetExceptionPolicyName()
{
    string actionName =
        ControllerContext.RouteData.GetRequiredString("action");
    ActionDescriptor actionDescriptor =
        this.Descriptor.FindAction(ControllerContext, actionName);
    if (null == actionDescriptor)
    {
        return string.Empty;
    }
    ExceptionPolicyAttribute exceptionPolicyAttribute =
        actionDescriptor.GetCustomAttributes(true)
            .OfType<ExceptionPolicyAttribute>().FirstOrDefault() ??
        Descriptor.GetCustomAttributes(true)
            .OfType<ExceptionPolicyAttribute>().FirstOrDefault() ??
        new ExceptionPolicyAttribute("");
    return exceptionPolicyAttribute.ExceptionPolicyName;
}

//获取 Handle-Error-Action 名称
public string GetHandleErrorActionName()
{
    string actionName =
        ControllerContext.RouteData.GetRequiredString("action");
    ActionDescriptor actionDescriptor =
        this.Descriptor.FindAction(ControllerContext, actionName);
    if (null == actionDescriptor)
    {
        return string.Empty;
    }
    HandleErrorActionAttribute handleErrorActionAttribute =
        actionDescriptor.GetCustomAttributes(true)
            .OfType<HandleErrorActionAttribute>().FirstOrDefault() ??
        Descriptor.GetCustomAttributes(true)
            .OfType<HandleErrorActionAttribute>().FirstOrDefault() ??
        new HandleErrorActionAttribute("");
    return handleErrorActionAttribute.HandleErrorAction;
}

//用于执行 Handle-Error-Action 的 ActionInvoker
public HandleErrorActionInvoker HandleErrorActionInvoker
{ get; private set; }

public ExtendedController()
{
    this.HandleErrorActionInvoker = new HandleErrorActionInvoker();
}
}

```

ExtendedController 的 Descriptor 属性返回描述自身的 ControllerDescriptor 对象，实际上是一个 ReflectedControllerDescriptor 对象。为了避免频繁的反射操作造成对性能的影响，我们将解析出来的 ReflectedControllerDescriptor 对象针对 Controller 类型进行了全局性缓存。

GetExceptionPolicyName 方法用于返回当前采用的异常处理策略名称。异常处理策略名

称是通过具有如下定义的 `ExceptionPolicyAttribute` 特性来指定的，该特性既可以应用在 `Controller` 类型上，也可以应用在 `Action` 方法上，换句话说，我们可以采用不同的策略来处理从不同 `Action` 执行过程中抛出的异常。`GetExceptionPolicyName` 方法利用 `ControllerDescriptor` 和 `ActionDescriptor` 可以很容易地得到应用的 `ExceptionPolicyAttribute` 特性，进而得到相应的异常处理策略名称。

```
[AttributeUsage( AttributeTargets.Class| AttributeTargets.Method,
    AllowMultiple = false, Inherited = true)]
public class ExceptionPolicyAttribute: Attribute
{
    public string ExceptionPolicyName { get; private set; }
    public ExceptionPolicyAttribute(string exceptionPolicyName)
    {
        this.ExceptionPolicyName = exceptionPolicyName;
    }
}
```

另一个方法 `GetHandleErrorActionName` 用于获取通过应用在 `Action` 方法上的特性 `HandleErrorActionAttribute` 设置的 `Handle-Error-Action` 的名称。该特性定义如下，它既可以应用于某个 `Action` 方法，也可以应用于 `Controller` 类。`GetHandleErrorActionName` 方法同样利用 `ControllerDescriptor` 和 `ActionDescriptor` 得到应用的 `ExceptionPolicyAttribute` 特性，并最终得到对应的异常处理 `Action` 名称。

```
[AttributeUsage( AttributeTargets.Class| AttributeTargets.Method,
    AllowMultiple = false)]
public class HandleErrorActionAttribute: Attribute
{
    public string HandleErrorAction { get; private set; }
    public HandleErrorActionAttribute(string handleErrorAction = "")
    {
        this.HandleErrorAction = handleErrorAction;
    }
}
```

通过 `HandleErrorActionAttribute` 特性设置的 `Handle-Error-Action` 需要手工执行以实现当前请求的响应，为此我们创建了一个具有如下定义的 `HandleErrorActionInvoker`。它是 `ControllerActionInvoker` 的子类，`Handle-Error-Action` 的执行以及对当前请求的响应实现在虚方法 `InvokeActionMethod` 中。`ExtendedController` 的 `HandleErrorActionInvoker` 返回的就是这样一个对象。

```
public class HandleErrorActionInvoker: ControllerActionInvoker
{
    public virtual ActionResult InvokeActionMethod(
        ControllerContext controllerContext, ActionDescriptor actionDescriptor)
    {
        IDictionary<string, object> parameterValues = this.GetParameterValues(
            controllerContext, actionDescriptor);
        return base.InvokeActionMethod(controllerContext, actionDescriptor,
            parameterValues);
    }
}
```

整个异常处理和最终对请求的响应实现在如下所示的 `OnException` 方法中，流程并不复杂，在这里就不再赘述了。

```
public class ExtendedController: Controller
{
    //其他成员
    protected override void OnException(ExceptionContext filterContext)
    {
        //或者当前的 ExceptionPolicy, 如果不存在, 则直接调用基类 OnException 方法
        string exceptionPolicyName = this.GetExceptionPolicyName();
        if (string.IsNullOrEmpty(exceptionPolicyName))
        {
            base.OnException(filterContext);
            return;
        }

        //利用 EntLib 的 EHAB 进行异常处理, 并获取错误消息和最后抛出的异常
        filterContext.ExceptionHandled = true;
        Exception exceptionToThrow;
        string errorMessage;
        try
        {
            ExceptionPolicy.HandleException(filterContext.Exception,
                exceptionPolicyName, out exceptionToThrow);
            errorMessage = System.Web.HttpContext.Current.GetErrorMessage();
        }
        finally
        {
            System.Web.HttpContext.Current.ClearErrorMessage();
        }
        exceptionToThrow = exceptionToThrow ?? filterContext.Exception;

        //对于 Ajax 请求, 直接返回一个用于封装异常的 JsonResult
        if (Request.IsAjaxRequest())
        {
            filterContext.Result = Json(new ExceptionDetail(
                exceptionToThrow, errorMessage));
            return;
        }

        //如果设置了匹配的 HandleErrorAction, 则调用之;
        //否则将 Error View 呈现出来
        string handleErrorAction = this.GetHandleErrorActionName();
        string controllerName =
            ControllerContext.RouteData.GetRequiredString("controller");
        string actionName =
            ControllerContext.RouteData.GetRequiredString("action");
        errorMessage = string.IsNullOrEmpty(errorMessage) ?
            exceptionToThrow.Message : errorMessage;
        if (string.IsNullOrEmpty(handleErrorAction))
        {
            filterContext.Result = View("Error",
                new ExtendedHandleErrorInfo(exceptionToThrow, controllerName,
                    actionName, errorMessage));
        }
        else
        {

```

```

        ActionDescriptor actionDescriptor = Descriptor.FindAction(
            ControllerContext, handleErrorAction);
        ModelState.AddModelError("", errorMessage);
        filterContext.Result = this.HandleErrorActionInvoker
            .InvokeActionMethod(ControllerContext, actionDescriptor);
    }
}

```

在调用 EntLib 的 EHAB 对异常处理过程中, 允许相应的 `ExceptionHandler` 设置一个友好的错误消息, 而这个消息被保存在当前 `HttpContext` 的 `Items` 中。在调用异常处理方法之前, 我们将错误消息添加到当前的 `ModelState` 中, 这也是为什么在上面的实例演示中错误消息会自动出现在 `ValidationSummary` 中的根本原因。

EHAB 完成了对异常的处理之后从调用 `HttpContext` 具有如下定义的扩展方法 `GetErrorMessage` 提取错误消息, 另一个扩展方法 `ClearErrorMessage` 方法实现对错误消息的清除。除了这两个扩展方法我们还定义了另一个用于设置错误消息的 `SetErrorMessage` 方法。

```

public static class HttpContextExtensions
{
    public static string keyOfErrorMessage = Guid.NewGuid().ToString();

    public static void SetErrorMessage(this HttpContext context,
        string errorMessage)
    {
        context.Items[keyOfErrorMessage]=errorMessage;
    }

    public static string GetErrorMessage(this HttpContext context)
    {
        return context.Items[keyOfErrorMessage] as string;
    }

    public static void ClearErrorMessage(this HttpContext context)
    {
        if (context.Items.Contains(keyOfErrorMessage))
        {
            context.Items.Remove(keyOfErrorMessage);
        }
    }
}

```

用于设置错误信息的 `ErrorMessageHandler` 以及对应配置元素类型 `ErrorMessageHandlerData` 定义如下。`ErrorMessageHandler` 表示错误消息的 `ErrorMessage` 属性在构造函数中被初始化, 而在实现的 `HandleException` 方法中直接通过调用当前 `HttpContext` 的扩展方法 `SetErrorMessage` 进行错误消息的设置。

```

[ConfigurationElementType(typeof(ErrorMessageHandlerData))]
public class ErrorMessageHandler: IExceptionHandler
{
    public string ErrorMessage { get; private set; }
    public ErrorMessageHandler(string errorMessage)
    {
        this.ErrorMessage = errorMessage;
    }
}

```

```

    }
    public Exception HandleException(Exception exception,
        Guid handlingInstanceId)
    {
        if (null != HttpContext.Current)
        {
            HttpContext.Current.SetErrorMessage(this.ErrorMessage);
        }
        return exception;
    }
}

public class ErrorMessageHandlerData : ExceptionHandlerData
{
    [ConfigurationProperty("errorMessage", IsRequired=true)]
    public string ErrorMessage
    {
        get { return (string)this["errorMessage"]; }
        set { this["errorMessage"] = value; }
    }
    public override IEnumerable<TypeRegistration> GetRegistrations(
        string namePrefix)
    {
        yield return new TypeRegistration<IExceptionHandler>(
            () => new ErrorMessageHandler(this.ErrorMessage))
        {
            Name = this.BuildName(namePrefix),
            Lifetime = TypeRegistrationLifetime.Transient
        };
    }
}

```

7.3.6 ResultFilter

如果 Action 方法返回一个 ActionResult 对象，ActionInvoker 在完成了 Action 方法的执行后会调用返回的 ActionResult 对象的 ExecuteResult 方法以实现对请求的响应。通过自定义 ActionFilter 我们不仅可以在 Action 方法执行前后完成一些额外的操作，甚至还可以通过 Action 方法执行前设置当前 ActionExecutingContext 的 Result 属性直接对请求作出响应。如果我们需要对 ActionResult 的执行进行类似的控制，可以自定义相应的 ResultFilter。

ResultFilter 实现了接口 System.Web.Mvc.IResultFilter。如下面的代码片段所示，IResultFilter 定义了两个方法 OnResultExecuting 和 OnResultExecuted，它们将在 ActionResult 执行前后被执行。两个方法各自具有一个基于上下文类型的参数，类型名称分别为 ResultExecutingContext 和 ResultExecutedContext。两个上下文类型均定义在 System.Web.Mvc 命名空间下，具体定义如下所示。

```

public interface IResultFilter
{
    void OnResultExecuted(ResultExecutedContext filterContext);
    void OnResultExecuting(ResultExecutingContext filterContext);
}

```

```

public class ResultExecutingContext : ControllerContext
{
    public ResultExecutingContext();
    public ResultExecutingContext(ControllerContext controllerContext,
        ActionResult result);

    public bool Cancel { get; set; }
    public virtual ActionResult Result { get; set; }
}

public class ResultExecutedContext : ControllerContext
{
    public ResultExecutedContext();
    public ResultExecutedContext(ControllerContext controllerContext,
        ActionResult result, bool canceled, Exception exception);

    public virtual bool Canceled { get; set; }
    public virtual Exception Exception { get; set; }
    public bool ExceptionHandled { get; set; }
    public virtual ActionResult Result { get; set; }
}

```

包含 `ActionResult` 的执行的整个 `ResultFilter` 链的执行流程与 `ActionFilter` 链的执行流程比较类似。在正常情况下 `ResultFilter` 链会按照类似于图 7-9 所示的流程执行（将 `ActionFilter` 和 `Action` 方法的执行分别看成是 `ResultFilter` 和 `ActionResult` 的执行）。

如果某个 `ResultFilter` 在执行 `OnResultExecuting` 过程将 `ResultExecutingContext` 的 `Cancel` 属性设置为 `True`，后续 `ResultFilter` 和最终的 `ActionResult` 都不会被执行，但是之前 `ResultFilter` 的 `OnResultExecuted` 方法会照常执行。

本章小结

为了改善 Web 应用的吞吐量、可用性和响应能力，ASP.NET 采用了线程池的机制来进行请求的处理。对于一些相对耗时的 I/O 绑定型操作，我们倾向于采用异步的方式来定义对应的 `Action`。ASP.NET MVC 4.0 保留了之前版本的异步 `Action` 定义方式（在继承自 `AsyncController` 的类型中以两个匹配的方法 `XxxAsync/XxxCompleted` 来定义），同时采用并行编程模式提供了一种全新的异步 `Action` 定义方式（将异步 `Action` 定义成一个返回类型为 `Task` 的方法）。在定义异步 `Action` 的时候，我们在异步操作完成之后需要利用 `AsyncManager` 向 ASP.NET MVC 发送通知。除此之外，异步操作向回调操作传递参数以及超时管理也可以通过它来完成。

就整个 ASP.NET MVC 进行请求处理的流程来看，很多处理环节都体现了同步和异步之间的差别，比如 `MvcHandler` 对请求的处理，`Controller` 和 `Action` 的执行。很多核心的组件都具有同步和异步两个版本，比如定义的 `Controller` 类型可以继承 `IController` 和 `IAsyncController` 接口；用于执行 `Action` 的 `ActionInvoker` 具有 `ControllerActionInvoker` 和 `AsyncControllerActionInvoker` 两种类型；它们创建的用于描述 `Controller` 的 `ControllerDescriptor`

的具体类型分别是 `ReflectedControllerDescriptor` 和 `ReflectedAsync ControllerDescriptor`；具体用于描述同步 `Action` 的 `ActionDescriptor` 类型是 `ReflectedAction Descriptor`，以不同方式定义的两类异步 `Action` 分别通过类型 `ReflectedAsyncActionDescriptor` 和 `TaskAsyncActionDescriptor` 来描述。

`ActionInvoker` 直接调用 `ActionDescriptor` 的 `Execute` 或者 `BeginExecute/EndExecute` 方法来执行目标 `Action` 方法，具体的 `Action` 方法执行通过反射来实现。伴随着目标 `Action` 方法执行的还有一系列筛选器的执行，ASP.NET MVC 定义了四种类型的筛选器（`AuthorizationFilter`、`ActionFilter`、`ExceptionHandler` 和 `ResultFilter`）。ASP.NET MVC 框架本身提供的一些功能是通过相应的筛选器来实现的，我们也可以根据需要自定义筛选器。

第 8 章 View 的呈现

定义在 Controller 中的 Action 方法一般会返回一个 ActionResult 对象对请求予以响应，ASP.NET MVC 定义了一系列 ActionResult 类型，合理地使用它们可以使请求响应变得非常容易。ReviewResult 是最为常用也是最为重要的 ActionResult，它利用可扩展的 View 引擎获取对应的 View 并最终将其呈现出来。

8.1 ActionResult

HTTP 是一个单纯采用请求/回复消息交换模式的网络协议, Web 服务器在接收并处理来自客户端的请求后会根据处理结果对请求予以响应。一般来说针对请求的处理最终体现在对目标 Action 方法的执行上, 我们可以在定义 Action 方法中人为地控制对请求的响应。如下面的代码片段所示, 抽象类 Controller 具有一个只读的 Response 属性表示当前的 HttpResponseMessage, 我们也可以间接地通过当前 HttpContext 或者 ControllerContext 来获取用于响应请求的当前 HttpResponseMessage 对象。

```
public abstract class Controller : ControllerBase, ...
{
    //其他成员
    public HttpResponseMessage Response { get; }
    public HttpContextBase HttpContext { get; }
}

public abstract class ControllerBase : IController
{
    //其他成员
    public ControllerContext ControllerContext { get; set; }
}
```

原则上任何类型的响应都可以利用当前 HttpResponseMessage 来实现, 但是我们一般并不这么做, 而是将针对请求的响应实现在一个 System.Web.Mvc.ActionResult 对象中。如下面的代码片段所示, ActionResult 是一个抽象类型, 请求响应实现在抽象 ExecuteResult 方法中。

```
public abstract class ActionResult
{
    //其他成员
    public abstract void ExecuteResult(ControllerContext context);
}
```

顾名思义, ActionResult 就是 Action 执行的结果。ActionInvoker 在完成对 Action 方法的执行后, 如果返回一个 ActionResult 对象, ActionInvoker 会将当前 ControllerContext 作为参数调用其 ExecuteResult 方法。View 的最终呈现是通过 ActionResult 的子类 ViewResult 来完成的, 除了 ViewResult, ASP.NET MVC 还为我们定义了一些额外的 ActionResult。

8.1.1 EmptyResult

上面我们谈到 Action 方法返回的 ActionResult 对象被 ActionInvoker 调用以实现当前请求的响应, 其实这种说法不够准确。不论 Action 方法是否具有返回值, 也不论它的返回值是什么类型, ActionInvoker 最终都会创建相应的 ActionResult 对象。如果 Action 方法返回类型为 void, 或者返回值为 Null, 最终生成的就是一个 System.Web.Mvc.EmptyResult 对象。如下面的代码片段所示, 在重写的 ExecuteResult 方法中 EmptyResult 其实什么都没有做, 所

以 `EmptyResult` 是一个“空”的 `ActionResult`。

```
public class EmptyResult : ActionResult
{
    public override void ExecuteResult(ControllerContext context)
    {
    }
}
```

`EmptyResult` 其实体现了一种设计思想。用于处理请求的 ASP.NET MVC 框架采用管道式设计，整个处理流程具有三个基本的环节，即“Action 方法的执行”、“生成 `ActionResult`”和“执行 `ActionResult`”。可能这个流程不适合某些特殊的请求，比如 `Action` 方法不具有返回值或者返回值为 `Null`，那么后面的两个环节可以忽略。为了让处理管道对所有的请求“一视同仁”，我们可以做一些适配工作，虽然 `EmptyResult` 什么都没有做，但是它却在上述这两种场景中起到了适配器的作用。

8.1.2 ContentResult

`System.Web.Mvc.ContentResult` 使 ASP.NET MVC 采用我们提供的内容来响应请求。如下面的代码片段所示，可以利用 `ContentResult` 的 `Content` 属性以字符串的形式指定响应的内容，另外两个属性 `ContentEncoding` 和 `ContentType` 则控制采用的字符编码方式和媒体类型（MIME 类型）。抽象类 `Controller` 定义了如下三个受保护的 `Content` 方法重载，可以调用它们根据指定的内容、编码和媒体类型创建相应的 `ContentResult`。

```
public class ContentResult : ActionResult
{
    public override void ExecuteResult(ControllerContext context);

    public string      Content { get; set; }
    public Encoding    ContentEncoding { get; set; }
    public string      ContentType { get; set; }
}

public abstract class Controller : ControllerBase, ...
{
    //其他成员
    protected ContentResult Content(string content);
    protected ContentResult Content(string content, string contentType);
    protected virtual ContentResult Content(string content,
        string contentType, Encoding contentEncoding);
}
```

在重写的 `ExecuteResult` 方法中，`ContentResult` 利用作为参数的 `ControllerContext` 对象得到当前的 `HttpResponse` 对象，并借助它将提供的内容按照希望的编码方式和媒体类型对请求予以响应，具体的实现如下面的代码片段所示。

```
public class ContentResult : ActionResult
{
    //其他成员
```

```

public override void ExecuteResult(ControllerContext context)
{
    HttpResponseBase response = context.HttpContext.Response;
    if (!string.IsNullOrEmpty(this.ContentType))
    {
        response.ContentType = this.ContentType;
    }
    if (this.ContentEncoding != null)
    {
        response.ContentEncoding = this.ContentEncoding;
    }
    if (this.Content != null)
    {
        response.Write(this.Content);
    }
}
}

```

上面我们说过, ASP.NET MVC 为了能够采用相同的流程来处理所有的请求, 不论 Action 是否具有返回值, 具有怎样的返回值, ActionInvoker 都会创建相应的 ActionResult。对于不具有返回值或者返回 Null 的 Action 方法调用来说, 最终创建的是一个 EmptyResult 对象, 那么如果返回值不是一个 ActionResult 对象, ActionInvoker 最终会创建怎样一个 ActionResult 对象呢?

如果 Action 方法执行后的返回值是一个 ActionResult, ActionInvoker 会直接利用它来进行请求响应, 否则它会将对象转换成字符串并以此创建一个 ContentResult 对象。ControllerActionInvoker 根据 Action 方法的返回值生成相应 ActionResult 对象的逻辑体现在它的 CreateActionResult 方法上。如下面的代码片段所示, 这是一个受保护的虚方法, 最后一个参数 (actionReturnValue) 表示执行 Action 方法得到的返回值。

```

public class ControllerActionInvoker : IActionInvoker
{
    //其他成员
    protected virtual ActionResult InvokeActionMethod(
        ControllerContext controllerContext, ActionDescriptor actionDescriptor,
        IDictionary<string, object> parameters);

    protected virtual ActionResult CreateActionResult(
        ControllerContext controllerContext,
        ActionDescriptor actionDescriptor, object actionReturnValue);
}

```

另一个受保护的 InvokeActionMethod 负责执行 Action 方法并返回相应的 ActionResult 对象, 它在执行 Action 方法后将得到的返回值作为参数调用 CreateActionResult 方法返回相应的 ActionResult 对象。

我们可以通过一个简单的实例来验证 ActionInvoker 根据 Action 方法返回值对 ActionResult 的创建逻辑。在一个 ASP.NET MVC 应用中我们定义了如下一个 HomeController, 其中定义了四个无参数的 Action 方法。Foo 返回一个具体的 ActionResult (RedirectResult) 对象, Bar 的返回类型为 void, Baz 返回值为 Null, 而 Qux 则返回一个 double 类型的数字。

```

public class HomeController : Controller
{
    //其他成员
    public ActionResult Foo()
    {
        return new RedirectResult("http://www.asp.net");
    }

    public void Bar()
    { }

    public ActionResult Baz()
    {
        return null;
    }

    public double Qux()
    {
        return 1.00;
    }
}

```

然后我们在 `HomeController` 中定义如下一个 `Action` 方法 `Index`。在该方法中，我们通过 `ActionInvoker` 属性得到当前的 `ActionInvoker` 对象，并以反射的方式调用其 `GetControllerDescriptor` 方法得到描述当前 `Controller` 的 `ControllerDescriptor` 对象。接下来我们调用这个 `ControllerDescriptor` 的 `FindAction` 方法得到描述定义在 `HomeController` 中的四个 `Action` (`Foo`、`Bar`、`Baz` 和 `Qux`) 的 `ActionDescriptor` 对象。针对每个具体的 `ActionDescriptor`，我们采用反射的方式调用其 `InvokeActionMethod` 方法得到最为 `Action` 方法返回值的 `ActionResult` 对象。我们将它们连同对应的 `ActionDescriptor` 对象构建一个 `Dictionary<ActionDescriptor, ActionResult>` 对象，并作为 `Model` 呈现在默认的 `View` 中。

```

public class HomeController : Controller
{
    //其他成员
    public ActionResult Index()
    {
        Dictionary<ActionDescriptor, ActionResult> actionResults =
            new Dictionary<ActionDescriptor, ActionResult>();
        MethodInfo getControllerDescriptor = this.ActionInvoker.GetType()
            .GetMethod("GetControllerDescriptor",
                BindingFlags.Instance | BindingFlags.NonPublic);
        ControllerDescriptor controllerDescriptor =
            (ControllerDescriptor)getControllerDescriptor
                .Invoke(this.ActionInvoker, new object[] { ControllerContext });
        MethodInfo invokeActionMethod = this.ActionInvoker.GetType()
            .GetMethod("InvokeActionMethod",
                BindingFlags.Instance | BindingFlags.NonPublic);

        string[] actions = new string[] { "Foo", "Bar", "Baz", "Qux" };
        Array.ForEach(actions, action =>
        {
            ActionDescriptor actionDescriptor = controllerDescriptor
                .FindAction(ControllerContext, action);
            ActionResult actionResult = (ActionResult)invokeActionMethod

```

```

        .Invoke(this.ActionInvoker, new object[] { ControllerContext,
            actionDescriptor, new Dictionary<string, object>() });
        actionResults.Add(actionDescriptor, actionResult);
    });
    return View(actionResults);
}
}

```

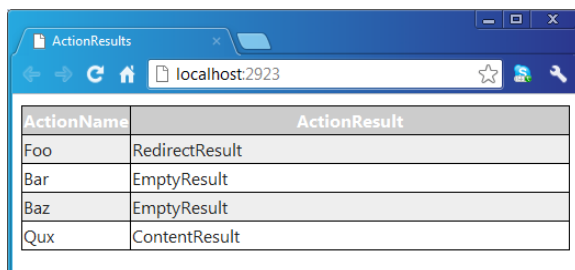
如下所示的是 Action 方法 Index 对应 View 的定义，IDictionary<ActionDescriptor, ActionResult>作为该 View 的 Model 类型。在该 View 中我们将存在于字典中的 ActionResult 对象的类型和对应的 Action 名称以表格的形式呈现出来。

```

@model IDictionary<ActionDescriptor, ActionResult>
<html>
<head>
<title>ActionResults</title>
</head>
<body>
<table>
<tr><th>ActionName</th><th>ActionResult</th></tr>
@foreach (var item in Model)
{
<tr>
<td>@item.Key.ActionName</td><td>@item.Value.GetType().Name</td>
</tr>
}
</table>
</body>
</html>

```

运行该程序后会在浏览器中得到如图 8-1 所示的输出结果，我们可以看到返回类型为 void 的 Action 方法 Bar 和返回值为 Null 的 Action 方法 Baz 执行后得到的都是一个 EmptyResult 对象，而返回非 ActionResult（double 类型）类型的 Action 方法 Qux 执行之后返回的是一个 ContentResult。（S801）



ActionName	ActionResult
Foo	RedirectResult
Bar	EmptyResult
Baz	EmptyResult
Qux	ContentResult

图 8-1 执行具有不同返回值的 Action 方法得到的 ActionResult 类型

实例演示：通过 ContentResult 实现主题定制（S802）

由于可以通过 ContentResult 的 ContentType 属性指定响应的媒体类型，所以我们不仅可以利用它来返回最终会在浏览器中显示的文本，还可以返回其他一些类型的内容，比如 JavaScript 脚本（“text/javascript”或者“application/javascript”）和 CSS 样式（“text/css”）等。

我们可以利用 `ContentResult` 实现“静态文本的动态响应”，也就是说我们可以在某个 `Action` 中根据当前的请求动态地生成一些文本（比如 CSS 样式），而这些文本内容在大部分情况下定义在静态文本文件中。

在接下来的这个实例演示中，我们将利用 `ContentResult` 实现对界面主题的定制。实现的机制非常简单，我们让一个返回类型为 `ContentResult` 的 `Action` 方法返回基于当前主题的 CSS 样式，而当前的主题通过一个可持久化的 `Cookie` 保存下来。我们在一个 ASP.NET MVC 应用中定义了如下一个 `HomeController`，其 `Action` 方法 `Css` 返回一个封装了 CSS 样式内容的 `ContentResult`。在该 `Action` 方法中，我们从请求中提取表示主题的 `Cookie`，并根据它生成基于当前主题的 CSS 样式（这里仅仅设置了字体类型和大小）。

```
public class HomeController : Controller
{
    //其他成员
    public ActionResult Css()
    {
        HttpCookie cookie = Request.Cookies["theme"]
            ?? new HttpCookie("theme", "default");
        switch (cookie.Value)
        {
            case "Theme1": return Content("body{font-family: SimHei;
                font-size:1.2em}", "text/css");
            case "Theme2": return Content("body{font-family: KaiTi;
                font-size:1.2em}", "text/css");
            default: return Content("body{font-family: SimSong;
                font-size:1.2em}", "text/css");
        }
    }
}
```

我们在 `HomeController` 中定义了如下两个 `Index` 方法。无参的 `Index` 方法（针对 HTTP-GET 请求）从预定义 `Cookie` 中提取当前的主题（如果没有则采用默认的主题“default”）并以 `ViewBag` 的形式传递给 `View`；另一个应用 `HttpPostAttribute` 特性的 `Index` 方法将通过参数指定的主题名称设置为响应的 `Cookie` 后，同样以 `ViewBag` 的形式保存当前的主题名称。两个 `Index` 方法最终都将默认的 `View` 呈现出来。

```
public class HomeController : Controller
{
    //其他成员
    public ActionResult Index()
    {
        HttpCookie cookie = Request.Cookies["theme"]
            ?? new HttpCookie("theme", "default");
        ViewBag.Theme = cookie.Value;
        return View();
    }

    [HttpPost]
    public ActionResult Index(string theme)
    {
        HttpCookie cookie = new HttpCookie("theme", theme);
```

```

        cookie.Expires = DateTime.MaxValue;
        Response.SetCookie(cookie);
        ViewBag.Theme = theme;
        return View();
    }
}

```

通过 `Css` 方法的定义可以看出我们定义了三个主题（`Theme1`、`Theme2` 和 `Default`），它们采用不同的中文字体（黑体、楷体 and 宋体）。`Action` 方法 `Index` 对应 `View` 具有如下一个表单，该表单中为这三个主题添加了相应的 `RadioButton` 使用户可以及时切换当前的主题。这个 `View` 最核心的部分是用于引用 `CSS` 文件的 `<link>` 元素，可以看到它的 `href` 属性指向的地址正对应着定义在 `HomeController` 中的 `Action` 方法 `Css`，也就是说最终用于控制页面样式的 `CSS` 是通过调用该 `Action` 获得的。

```

<html>
<head>
    <title>主题设置</title>
    <link type="text/css" rel="Stylesheet" href="@Url.Action("Css")" />
</head>
<body>
    @using(Html.BeginForm())
    {
        string theme = ViewBag.Theme.ToString();
        @Html.RadioButton("theme", "Default", theme == "Default")
        <span>默认主题（宋体）</span><br/>
        @Html.RadioButton("theme", "Theme1", theme == "Theme1")
        <span>主题 1（黑体）</span><br/>
        @Html.RadioButton("theme", "Theme2", theme == "Theme2")
        <span>主题 2（楷体）</span><br />
        <input type="submit" value="保存" />
    }
</body>
</html>

```

现在直接运行我们的程序并在出现的“主题设置”界面中设置不同的主题，界面的样式（字体）将会根据选择的主题而及时改变，具体的显示效果如图 8-2 所示。

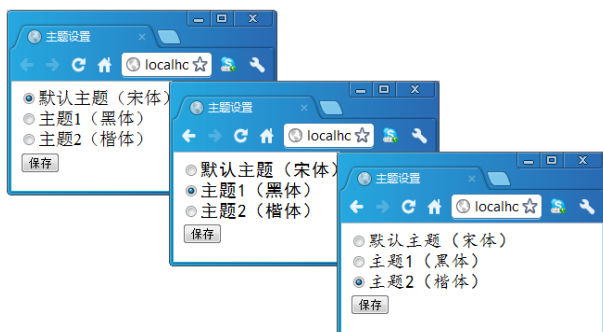


图 8-2 通过 `ContentResult` 实现的主题定制

8.1.3 FileResult

`System.Web.Mvc.FileResult` 是一个基于文件的 `ActionResult`，利用 `FileResult` 我们可以很容易地将某个物理文件的内容响应给客户端。如下面的代码片段所示，`FileResult` 具有一个表示媒体类型的只读属性 `ContentType`，该属性在构造函数中被初始化。当我们基于某个物理文件创建相应的 `FileResult` 对象的时候应该根据文件的类型指定媒体类型，比如说目标文件是一个 .jpg 图片，那么对应的媒体类型为 “image/jpeg”；对于一个 .pdf 文件，则采用 “application/pdf”。

```
public abstract class FileResult : ActionResult
{
    protected FileResult(string contentType);
    public override void ExecuteResult(ControllerContext context);
    protected abstract void WriteFile(HttpResponseBase response);

    public string ContentType { get; }
    public string FileNameDownloadName { get; set; }
}
```

针对文件的响应具有两种形式，即内联（Inline）和附件（Attachment）。一般来说，前者会利用浏览器直接打开响应的文件，而后者会以独立的文件下载到客户端。对于后者，我们一般会为下载的文件指定一个文件名，这个文件名可以通过 `FileResult` 的 `FileNameDownloadName` 属性来指定。文件响应在默认情况下采用内联的方式，如果需要采用附件的形式，需要为响应创建一个名称为 “Content-Disposition” 的报头，该报头值的格式为 “attachment; filename={FileNameDownloadName}”。

`FileResult` 仅仅是一个抽象类，文件内容的输出实现在抽象方法 `WriteFile` 中，该方法会在重写的 `ExecuteResult` 方法中调用。如果 `FileNameDownloadName` 属性不为空，意味着会采用附件的形式进行文件响应，`FileResult` 会在重写的 `ExecuteResult` 方法中进行 “Content-Disposition” 响应报头的设置。下面的代码片段基本上体现了 `ExecuteResult` 方法在 `FileResult` 中的实现。

```
public abstract class FileResult : ActionResult
{
    //其他成员
    public override void ExecuteResult(ControllerContext context)
    {
        HttpResponseBase response = context.HttpContext.Response;
        response.ContentType = this.ContentType;
        if (!string.IsNullOrEmpty(this.FileNameDownloadName))
        {
            //生成 Content-Disposition 响应报头值
            string headerValue =
                ContentDispositionUtil.GetHeaderValue(this.FileNameDownloadName);
            context.HttpContext.Response.AddHeader("Content-Disposition",
                headerValue);
        }
        this.WriteFile(response);
    }
}
```

ASP.NET MVC 定义了三个具体的 `FileResult`，分别是 `FileContentResult`、`FilePathResult` 和 `FileStreamResult`，接下来对它们进行单独介绍。

FileContentResult

`System.Web.Mvc.FileContentResult` 是针对文件内容创建的 `FileResult`。如下面的代码片段所示，`FileContentResult` 具有一个字节数组类型的只读属性 `FileContents` 表示响应文件的内容，该属性在构造函数中指定。`FileContentResult` 针对文件内容的响应实现也很简单，从如下所示的 `WriteFile` 方法定义可以看出，它先获得当前 `HttpResponse` 的 `OutputStream` 属性表示的输出流，然后调用其 `Write` 方法直接将表示文件内容的字节数组进行输出。

```
public class FileContentResult : FileResult
{
    public byte[] FileContents { get; }
    public FileContentResult(byte[] fileContents, string contentType) ;

    protected override void WriteFile(HttpResponseBase response)
    {
        response.OutputStream.Write(this.FileContents, 0,
            this.FileContents.Length);
    }
}

public abstract class Controller : ControllerBase, ...
{
    // 其他成员
    protected FileContentResult File(byte[] fileContents,
        string contentType);
    protected virtual FileContentResult File(byte[] fileContents,
        string contentType, string fileName);
}
```

抽象类 `Controller` 中定义了如上两个 `File` 重载，它们根据指定的字节数组、媒体类型和下载文件名（可选）生成相应的 `FileContentResult`。由于 `FileContentResult` 是根据字节数组创建的，当我们需要动态生成响应文件内容（而不是从物理文件中读取）时，`FileContentResult` 是一个不错的选择。

FilePathResult

从名称可以看出，`System.Web.Mvc.FilePathResult` 是一个根据物理文件路径创建的 `FileResult`。如下面的代码片段所示，表示响应文件的路径通过只读属性 `FileName` 表示，该属性在构造函数中被初始化。在实现的 `WriteFile` 方法中，它直接将文件路径作为参数调用当前 `HttpResponse` 的 `TransmitFile` 方法实现了目标文件内容的输出。抽象类 `Controller` 同样定义了两个 `File` 方法重载来根据文件路径创建相应的 `FilePathResult`。

```
public class FilePathResult : FileResult
{
    public string FileName { get; }
```

```

    public FilePathResult(string fileName, string contentType);

    protected override void WriteFile(HttpResponseBase response)
    {
        response.TransmitFile(this.FileName);
    }
}

public abstract class Controller : ControllerBase, ...
{
    //其他成员
    protected FilePathResult File(string fileName, string contentType);
    protected virtual FilePathResult File(string fileName,
        string contentType, string fileNameDownloadName);
}

```

FileStreamResult

`System.Web.Mvc.FileStreamResult` 允许我们通过一个用于读取文件内容的 `Stream` 对象来创建 `FileResult`。如下面的代码片段所示，读取文件的 `Stream` 对象通过只读属性 `FileStream` 表示，该属性在构造函数中被初始化。在实现的 `WriteFile` 方法中，它通过指定的文件流读取文件内容，并最终调用当前 `HttpResponse` 的 `OutputStream` 属性的 `Write` 方法将读取的内容写入当前 HTTP 响应的输出流中。抽象类 `Controller` 中同样定义了两个 `File` 方法，它们重载根据文件读取的 `Stream` 对象创建相应的 `FileStreamResult`。

```

public class FileStreamResult : FileResult
{
    public Stream FileStream { get; }
    public FileStreamResult(Stream fileStream, string contentType);

    protected override void WriteFile(HttpResponseBase response)
    {
        Stream outputStream = response.OutputStream;
        using (this.FileStream)
        {
            byte[] buffer = new byte[0x1000];
            while (true)
            {
                int count = this.FileStream.Read(buffer, 0, 0x1000);
                if (count == 0)
                {
                    return;
                }
                outputStream.Write(buffer, 0, count);
            }
        }
    }
}

public abstract class Controller : ControllerBase, ...
{

```

```
//其他成员
protected FileStreamResult File(Stream fileStream,
    string contentType);
protected virtual FileStreamResult File(Stream fileStream,
    string contentType, string fileName);
}
```

实例演示：通过 FileResult 发布图片（S803）

为了让读者对 `FileResult` 具有更加深刻的认识，我们通过一个实例来演示如何通过 `FileResult` 来对外发布图片。在一个 ASP.NET MVC 应用的根目录下添加一个名为 `images` 的子目录来存放发布的.jpg 图片，然后定义如下一个 `HomeController`。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    public ActionResult Image(string id)
    {
        string path = Server.MapPath("/images/" + id + ".jpg");
        return File(path, "image/jpeg");
    }
}
```

图片的发布体现在 `Action` 方法 `Image` 上，表示图片 ID 的参数同时作为图片的文件名（不含扩展名）。在该方法中，我们根据图片 ID 解析出对应文件的路径后，直接调用 `File` 方法创建一个媒体类型为“image/jpeg”的 `FilePathResult`。

如下所示的是 `Action` 方法 `Index` 对应 `View` 的定义，在该 `View` 中我们通过一个列表显示 6 张图片。对于显示图片的 `` 元素的 `src` 属性来说，它正是指向定义在 `HomeController` 的 `Action` 方法 `Image`，而指定的表示图片 ID 的参数分别是 001、002、…、006。

```
<html>
  <head>
    <title>Gallery</title>
    <style type="text/css">
      li{list-style-type:none; float:left; margin:10px 10px 0px 0px;}
      img{width:100px; height:100px;}
    </style>
  </head>
  <body>
    <ul>
      <li>
        
      </li>
      <li>
        
      </li>
      <li>
        
      </li>
    </ul>
  </body>
</html>
```

```

        <li>
            
        </li>
        <li>
            
        </li>
        <li>
            
        </li>
    </ul>
</body>
</html>

```

我们将 6 张.jpg 图片存放到“/images”目录下，并分别命名为 001、002、…、006。直接运行程序之后这 6 张图片会以如图 8-3 所示的效果显示在浏览器上。

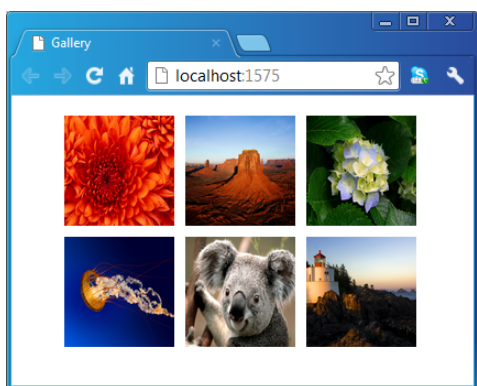


图 8-3 通过返回类型为 `FileResult` 的 `Action` 发布的图片

8.1.4 JavaScriptResult

`System.Web.Mvc.JavaScriptResult` 使我们可以在服务端动态地生成一段 JavaScript 脚本，并以此作为请求的响应，而这段脚本会在客户端被执行。其实 `JavaScriptResult` 的实现非常简单，它仅仅是将表示 JavaScript 脚本的字符串通过当前的 `HttpResponse` 响应给请求的客户端而已。如下面的代码片段所示，`JavaScriptResult` 的属性 `Script` 表示响应的 JavaScript 脚本，而用于响应 JavaScript 脚本的 `ExecuteResult` 方法除了将脚本内容写入当前 `HttpResponse` 之外，还会将响应的媒体类型设置为“`application/x-javascript`”（不是“`text/javascript`”）。

```

public class JavaScriptResult : ActionResult
{
    public override void ExecuteResult(ControllerContext context)
    {
        HttpResponseBase response = context.HttpContext.Response;
        response.ContentType = "application/x-javascript";
        response.Write(this.Script);
    }
    public string Script { get; set; }
}

```

```
public abstract class Controller : ControllerBase, ...
{
    //其他成员
    protected virtual JavaScriptResult JavaScript(string script);
}
```

抽象类 `Controller` 中定义了如上一个 `JavaScript` 方法，它根据指定的脚本字符串创建一个 `JavaScriptResult`。实际上我们完全可以通过 `ContentResult` 来实现与 `JavaScriptResult` 一样的脚本响应功能，下面的两段程序是等效的。大部分浏览器会将媒体类型“`application/x-javascript`”等同于“`text/javascript`”，所以在通过 `ContentResult` 进行脚本响应时将媒体类型设置为“`text/javascript`”可以起到相同的效果。返回类型为 `JavaScriptResult` 的 `Action` 方法一般用于处理 `Ajax` 请求。

```
//JavaScriptResult:
public class FooController : Controller
{
    public ActionResult JavaScript()
    {
        return JavaScript("alert('Hello World!');");
    }
}

//ContentResult:
public class FooController : Controller
{
    public ActionResult JavaScript()
    {
        return Content("alert('Hello World!');", "application/x-javascript");
    }
}
```

我们照例演示一个通过 `JavaScriptResult` 进行脚本响应的例子，我们演示一个在线购物的场景：用户完成了商品选购之后提交订单，服务端在处理订单的时候需要确认订购的商品是否超出了对应的库存量，如果存量充裕则正常处理该订单，否则提示库存不足，并将商品实时库存量显示给用户让他修正相应商品的购买量。我们利用 `JavaScript` 的方式来提示订单处理结果的消息（成功处理或者库存不足），很显然这段 `JavaScript` 的内容应该是动态的（库存量是动态的）。

我们在一个 `ASP.NET MVC` 应用中定义一个 `ShoppingCart` 类表示购物车。如下面的代码片段所示，`ShoppingCart` 是表示购物车商品项 `ShoppingCartItem` 对象的列表，而 `ShoppingCartItem` 的三个属性（`Id`、`Name` 和 `Quantity`）分别表示商品 ID、名称和订购数量。

```
public class ShoppingCart : List<ShoppingCartItem>
{
}

public class ShoppingCartItem
{
    public string Id { get; set; }
    public string Name { get; set; }
    public int Quantity { get; set; }
}
```

然后我们创建如下一个 HomeController，在默认的动作方法 Index 中创建一个包含三个商品的 ShoppingCart 对象，并将其作为 Model 呈现在对应的 View 中。Action 方法 ProcessOrder 用于处理提交的购买订单，如果订购商品的数量没有超过库存量（通过一个静态字典字段 stock 表示），则调用 alert 函数提示“购物订单成功处理”，否则提示“库存不足”，并将相应商品当前库存量显示出来。

```
public class HomeController : Controller
{
    private static Dictionary<string, int> stock = new Dictionary<string, int>();

    static HomeController()
    {
        stock.Add("001", 20);
        stock.Add("002", 30);
        stock.Add("003", 40);
    }

    public ActionResult Index()
    {
        ShoppingCart cart = new ShoppingCart();
        cart.Add(new ShoppingCartItem { Id = "001", Quantity=1,
            Name = "商品 A" });
        cart.Add(new ShoppingCartItem { Id = "002", Quantity = 1,
            Name = "商品 B" });
        cart.Add(new ShoppingCartItem { Id = "003", Quantity = 1,
            Name = "商品 C" });
        return View(cart);
    }

    public ActionResult ProcessOrder(ShoppingCart cart)
    {
        StringBuilder sb = new StringBuilder();
        foreach (var cartItem in cart)
        {
            if (!CheckStock(cartItem.Id, cartItem.Quantity))
            {
                sb.Append(string.Format("{0}: {1};",
                    cartItem.Name, stock[cartItem.Id]));
            }
        }
        if (string.IsNullOrEmpty(sb.ToString()))
        {
            return Content("alert('购物订单成功处理! ');", "text/javascript");
        }
        string script = string.Format("alert('库存不足! ({0})');",
            sb.ToString().TrimEnd(';'));
        return JavaScript(script);
    }

    private bool CheckStock(string id, int quantity)
    {
        return stock[id] >= quantity;
    }
}
```

如下所示的是 Action 方法 Index 对应 View 的定义,这是一个 Model 类型为 ShoppingCart 的强类型 View。在一个以 Ajax 请求提交的表单(表单的 Action 属性对应着上面定义的动作方法 ProcessOrder)中显示了购物车中的商品和数量,用户可以修改订购数量并通过点击“提交订单”按钮以 Ajax 请求的方式提交订单。

```
@model ShoppingCart
<html>
  <head>
    <title>用户登录</title>
    <script type="text/javascript" src="@Url.Content(
      "~/Scripts/jquery-1.6.2.js")"></script>
    <script type="text/javascript" src="@Url.Content(
      "~/Scripts/jquery.unobtrusive-ajax.js")"></script>
  </head>
  <body>
    @using (Ajax.BeginForm("ProcessOrder", new AjaxOptions()))
    {
      for (int i = 0; i < Model.Count; i++)
      {
        <div>
          @Html.HiddenFor(m=>m[i].Id)
          @Html.HiddenFor(m => m[i].Name)

          @Html.DisplayFor(m => m[i].Name):
          @Html.EditorFor(m => m[i].Quantity)
        </div>
      }
      <input type="submit" value="提交订单" />
    }
  </body>
</html>
```

运行程序后,一个包含三个商品的购物车信息会被呈现出来,当我们输入相应的订购数量并点击“提交订单”后,订单处理结果消息会弹出来。图 8-4 所示的就是库存不足的情况下显示的消息。(S804)

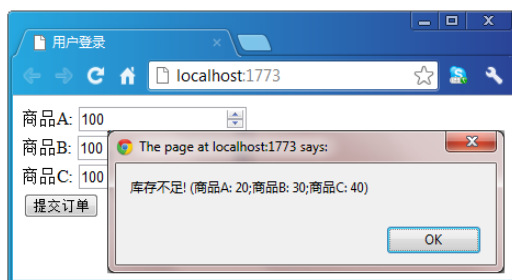


图 8-4 通过 JavaScriptResult 显示的提示消息

8.1.5 JsonResult

JavaScript 已经在 Web 应用中得到广泛的应用,而 JSON 则成了标准的数据格式。但是

对于后台程序来说，数据却是通过一个基于某种 CLR 类型的对象来承载的，当客户端调用某个 Action 方法并希望以 JSON 的格式返回请求的数据时，ASP.NET MVC 需要有一种机制将 CLR 对象转换成 JSON 格式予以响应，而这可以通过 `System.Web.Mvc.JsonResult` 来解决。

如下面的代码片段所示，`JsonResult` 具有一个 `object` 类型的属性 `Data` 表示需要被转换成 JSON 格式的数据对象。属性 `ContentEncoding` 和 `ContentType` 表示为当前响应设置的编码方式和媒体类型，默认采用的媒体类型为 “application/json”。

```
public class JsonResult : ActionResult
{
    public override void ExecuteResult(ControllerContext context);

    public object Data { get; set; }
    public Encoding ContentEncoding { get; set; }
    public string ContentType { get; set; }
    public JsonRequestBehavior JsonRequestBehavior { get; set; }
    public int? MaxJsonLength { get; set; }
    public int? RecursionLimit { get; set; }
}

public enum JsonRequestBehavior
{
    AllowGet,
    DenyGet
}
```

出于对安全的考虑，`JsonResult` 在默认的情况下不能作为对 HTTP-GET 请求的响应，在这种情况下并会直接抛出一个 `InvalidOperationException` 异常。我们可以通过它的 `JsonRequestBehavior` 属性开启对 HTTP-GET 请求的支持。该属性类型为 `System.Web.Mvc.JsonRequestBehavior` 枚举，两个枚举项 `AllowGet` 和 `DenyGet` 分别表示允许/拒绝支持对 HTTP-GET 请求的响应。`JsonResult` 的 `JsonRequestBehavior` 属性在初始化的时候被设置为 `DenyGet`，如果我们需要用创建的 `JsonResult` 来响应 HTTP-GET 请求，需要显式地将它的 `JsonRequestBehavior` 属性设置为 `AllowGet`。

CLR 对象到 JSON 格式字符串的序列化过程通过具有如下定义的序列化器 `System.Web.Script.Serialization.JavaScriptSerializer` 来完成。`JavaScriptSerializer` 的 `Serialize` 和 `Deserialize` 方法实现了 CLR 对象的序列化和对 JSON 字符串的反序列化。

```
public class JavaScriptSerializer
{
    //其他成员
    public string Serialize(object obj);
    public object Deserialize(string input, Type targetType);

    public int MaxJsonLength { get; set; }
    public int RecursionLimit { get; set; }
}
```

JavaScriptSerializer 具有两个整型的属性 MaxJsonLength 和 RecursionLimit，它们对应着 JsonResult 的同名属性。MaxJsonLength 限制了被反序列化和序列化生成的 JSON 字符串的长度，默认值为 2097152 (0x200000，等同于 4 MB 的 Unicode 字符串数据)。RecursionLimit 用于设置被序列化对象和反序列化生成对象结构的允许的层级数，默认值为 100。

定义在 JsonResult 的 ExecuteResult 方法通过 JavaScriptSerializer 对数据对象的序列化，并将序列化生成的 JSON 字符串作为内容对请求进行响应，具体的逻辑基本上可以通过下面的代码片段来体现。

```
public class JsonResult : ActionResult
{
    //其他成员
    public override void ExecuteResult(ControllerContext context)
    {
        //确认是否用于响应 HTTP-GET 请求
        if (this.JsonRequestBehavior == JsonRequestBehavior.DenyGet &&
            string.Compare(context.HttpContext.Request.HttpMethod, "GET", true)
                == 0)
        {
            throw new InvalidOperationException();
        }

        HttpResponseBase response = context.HttpContext.Response
        //设置媒体类型和编码方式
        response.ContentType = string.IsNullOrEmpty(this.ContentType) ?
            "application/json" : this.ContentType;
        if (this.ContentEncoding != null)
        {
            response.ContentEncoding = this.ContentEncoding;
        }

        //创建 JavaScriptSerializer 将数据对象序列化成 JSON 字符串并写入当前 HttpResponse
        if (null == this.Data) return;
        JavaScriptSerializer serializer = new JavaScriptSerializer()
        {
            MaxJsonLength = this.MaxJsonLength.HasValue ?
                this.MaxJsonLength.Value : 0x200000,
            RecursionLimit = this.RecursionLimit.HasValue ?
                this.RecursionLimit.Value : 100
        };
        response.Write(serializer.Serialize(this.Data));
    }
}
```

抽象类 Controller 同样定义了如下一系列的 Json 方法用于根据指定的数据对象、编码方式以及 JsonRequestBehavior 来创建相应的 JsonResult。

```
public abstract class Controller : ControllerBase, ...
{
    //其他成员
    protected internal JsonResult Json(object data);
    protected internal JsonResult Json(object data, string contentType);
    protected internal JsonResult Json(object data,
```

```

        JsonRequestBehavior behavior);
protected internal virtual JsonResult Json(object data, string contentType,
    Encoding contentEncoding);
protected internal JsonResult Json(object data, string contentType,
    JsonRequestBehavior behavior);
protected internal virtual JsonResult Json(object data, string contentType,
    Encoding contentEncoding, JsonRequestBehavior behavior);
}

```

8.1.6 HttpStatusCodeResult

每一个 HTTP 响应均具有一个表示响应状态的代码和一个可选的状态描述，正常情况下返回“200 OK”。System.Web.Mvc.HttpStatusCodeResult 使我们很容易地响应一个指定状态的回复。如下面的代码片段所示，HttpStatusCodeResult 具有 StatusCode 和 StatusDescription 两个只读的属性，它们分别表示响应状态码和状态描述信息。HttpStatusCodeResult 的 HTTP 状态既可以在构造函数中通过一个整数来指定，也可以通过 System.Net.HttpStatusCode 枚举形式来指定状态码。

```

public class HttpStatusCodeResult : ActionResult
{
    public HttpStatusCodeResult(int statusCode);
    public HttpStatusCodeResult(HttpStatusCode statusCode);
    public HttpStatusCodeResult(int statusCode, string statusDescription);
    public HttpStatusCodeResult(HttpStatusCode statusCode,
        string statusDescription);

    public override void ExecuteResult(ControllerContext context);

    public int        StatusCode { get; }
    public string      StatusDescription { get; }
}

```

HttpStatusCodeResult 实现在 ExecuteResult 方法中的请求响应逻辑很简单。如下面的代码片段所示，它仅仅是设置了当前 HttpResponse 的 StatusCode 和 StatusDescription 而已。值得一提的是，如果我们采用 Visual StudioDevelopment Server 作为 Web 应用的宿主，通过 HttpStatusCodeResult 的 StatusDescription 属性设置的状态描述信息不会反映在 HTTP 响应中，只有采用 IIS 作为宿主才会真正将此信息写入响应消息。

```

public class HttpStatusCodeResult : ActionResult
{
    //其他成员
    public override void ExecuteResult(ControllerContext context)
    {
        context.HttpContext.Response.StatusCode = this.StatusCode;
        if (this.StatusDescription != null)
        {
            context.HttpContext.Response.StatusDescription =
                this.StatusDescription;
        }
    }
}

```

```

    }
}

```

`HttpStatusCodeResult` 具有两个子类，一个是基于响应状态“404, Not Found”的 `System.Web.Mvc.HttpNotFoundResult`，另一个是基于响应状态“401, Not Authorized”的 `System.Web.Mvc.HttpUnauthorizedResult`，第7章“Action的执行”中筛选器 `AuthorizeAttribute` 在授权检验失败的情况下返回的就是一个 `HttpUnauthorizedResult` 对象。

8.1.7 RedirectResult/RedirectToRouteResult

`System.Web.Mvc.RedirectResult` 帮助我们实现针对某个地址的重定向，其作用与调用 `HttpResponse` 的 `Redirect/RedirectPermanent` 方法完全一致。如下面的代码片段所示，`RedirectResult` 具有两个只读属性 `Permanent` 和 `Url`，前者表示采用永久重定向还是暂时重定向，默认值为 `False`，后者表示重定向的目标地址，既可以采用绝对地址（比如 `http://www.asp.net`），也可以采用相对地址（比如 `~/account/register`）。

```

public class RedirectResult : ActionResult
{
    public RedirectResult(string url);
    public RedirectResult(string url, bool permanent);
    public override void ExecuteResult(ControllerContext context);

    public bool        Permanent { get; }
    public string      Url { get; }
}

```

暂时重定向和永久重定向可以分别通过调用 `HttpResponse` 的 `Redirect` 和 `RedirectPermanent` 方法来实现，实际上 `RedirectResult` 基于重定向的实现就是通过调用这两个方法来完成的，这可以通过如下所示的 `ExecuteResult` 方法的定义看出来。

```

public class RedirectResult : ActionResult
{
    //其他成员
    public override void ExecuteResult(ControllerContext context)
    {
        //其他操作
        string url = UrlHelper.GenerateContentUrl(this.Url, context.HttpContext);
        if (this.Permanent)
        {
            bool endResponse = false;
            context.HttpContext.Response.RedirectPermanent(url, false);
        }
        else
        {
            bool flag2 = false;
            context.HttpContext.Response.Redirect(url, false);
        }
    }
}

```

`RedirectResult` 使我们可以直接重定向到指定的目标地址，另一个类似的 `System.Web.Mvc.RedirectToRouteResult` 帮助我们根据注册的路由进行重定向。如下面的代码片段所示，`RedirectToRouteResult` 没有了表示重定向目标地址的 `Url` 属性，取而代之的是表示路由注册名称和路由参数的 `RouteName` 和 `RouteValues` 属性，ASP.NET MVC 根据这两个属性利用注册的路由解析出具体的重定向地址。

```
public class RedirectToRouteResult : ActionResult
{
    public RedirectToRouteResult(RouteValueDictionary routeValues);
    public RedirectToRouteResult(string routeName,
        RouteValueDictionary routeValues);
    public RedirectToRouteResult(string routeName,
        RouteValueDictionary routeValues, bool permanent);
    public override void ExecuteResult(ControllerContext context);

    public bool Permanent { get; }
    public string RouteName { get; }
    public RouteValueDictionary RouteValues { get; }
}
```

抽象类 `Controller` 中定义了一系列创建 `RedirectResult`/`RedirectToRouteResult` 的方法，比如 `Redirect`/`RedirectPermanent` 方法用于创建重定向到指定 URL 的 `RedirectResult`，`RedirectToAction`/`RedirectToActionPermanent` 用于创建重定向到指定的目标 Action 的 `RedirectResult`/`RedirectToRouteResult`，而 `RedirectToRoute`/`RedirectToRoutePermanent` 创建的 `RedirectResult`/`RedirectToRouteResult` 对象是针对注册的某个路由的。

暂时重定向和永久重定向有时又被称为“302 重定向”和“301 重定向”，302 和 301 表示响应的状态码。当我们调用 `HttpResponse` 的 `Redirect`/`RedirectPermanent` 方法时，除了会设置相应的响应状态码之外，还会将重定向的目标地址写入响应报头（`Location`），浏览器在接收到响应之后自动发起针对重定向目标地址的访问。

```
public class HomeController : Controller
{
    public ActionResult Redirect()
    {
        return Redirect("http://www.asp.net");
    }

    public ActionResult RedirectPermanent()
    {
        return RedirectPermanent("http://www.asp.net");
    }
}
```

在上面的代码片段中，我们定义了采用暂时重定向和永久重定向的 Action 方法 `Redirect` 和 `RedirectPermanent`，如果我们通过浏览器分别对它们发起访问，会得到具有如下内容的两个响应。两种重定向的不同作用主要体现在 SEO（Search engine optimization）上，搜索引擎会使用永久重定向目标地址更新自己的索引，对于暂时重定向则不会。

```
//1. Redirect
HTTP/1.1 302 Found
Server: ASP.NET Development Server/10.0.0.0
Date: Wed, 13 Jun 2012 09:34:15 GMT
X-AspNet-Version: 4.0.30319
X-AspNetMvc-Version: 4.0
Location: http://www.asp.net
Cache-Control: private
Content-Type: text/html; charset=utf-8
Content-Length: 135
Connection: Close

<html><head><title>Object moved</title></head><body><h2>Object moved to <a
href="http://www.asp.net">here</a>.</h2></body></html>

//2. RedirectPermanent
HTTP/1.1 301 Moved Permanently
Server: ASP.NET Development Server/10.0.0.0
Date: Wed, 13 Jun 2012 09:34:40 GMT
X-AspNet-Version: 4.0.30319
X-AspNetMvc-Version: 4.0
Location: http://www.asp.net
Cache-Control: private
Content-Type: text/html; charset=utf-8
Content-Length: 135
Connection: Close

<html><head><title>Object moved</title></head><body><h2>Object moved to <a
href="http://www.asp.net">here</a>.</h2></body></html>
```

8.2 ViewResult 与 ViewEngine

前面我们讨论了各种 `ActionResult`，与这些采用简单而直接的请求响应机制的 `ActionResult` 相比，`ViewResult` 基于 `View` 呈现的请求响应机制要复杂得多，它内部借助了 ASP.NET MVC 提供的 `View` 引擎实现了对 `View` 获取、激活和呈现。

8.2.1 View 引擎中的 View

ASP.NET MVC 为我们提供了两种 `View` 引擎，一种是传统的 `Web Form` 引擎（由于该引擎下 `View` 的设计与我们定义 `.aspx` 页面一致，又称为 `ASPX` 引擎），另外一种则是本书默认采用同时也是推荐使用的 `Razor` 引擎。在讨论两种 `View` 引擎的运行机制之前，我们有必要回答这么一个问题：“`View` 如何表示？”

提到 `View`，很多开发人员可能首先想到的是定义 `UI` 界面的 `.aspx`（`Web Form` 引擎）或者 `.cshtml/.vbhtml` 文件（`Razor` 引擎），其实对于 `View` 引擎来说，`View` 通过 `System.Web.Mvc.IView` 接口来表示。如下面的代码片段所示，`IView` 仅仅具有唯一的 `Render` 方法实现对 `View` 的呈现。

```

public interface IView
{
    void Render(ViewContext viewContext, TextWriter writer);
}

public class ViewContext : ControllerBase
{
    //其他成员
    public virtual bool ClientValidationEnabled { get; set; }
    public virtual bool UnobtrusiveJavaScriptEnabled { get; set; }

    public virtual TempDataDictionary TempData { get; set; }
    [Dynamic]
    public object ViewBag { [return: Dynamic] get; }
    public virtual ViewDataDictionary ViewData { get; set; }
    public virtual IView View { get; set; }
    public virtual TextWriter Writer { get; set; }
}

public abstract class HttpResponseMessage
{
    //其他成员
    public virtual TextWriter Output { get; set; }
}

```

IView 用于呈现 View 的 Render 方法具有两个参数，其中之一就是表示 View 上下文的 System.Web.Mvc.ViewContext 对象。通过上面的代码片段可以看出 ViewContext 是 ControllerBase 的子类，用于表示状态数据的 ViewData、ViewBag 和 TempData 对应着 ControllerBase 的同名属性。

ViewContext 具有两个布尔类型属性 ClientValidationEnabled 和 UnobtrusiveJavaScriptEnabled，它们分别表示是否支持客户端验证和 UnobtrusiveJavaScript。这两个属性可以通过如下所示的同名的 AppSettings 配置项进行设置。如果不具有对应的配置，两个属性默认值为 False。

```

<configuration>
  <appSettings>
    <add key="ClientValidationEnabled" value="true"/>
    <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
  </appSettings>
</configuration>

```

配置的范围是针对整个 Web 应用而言的，这个全局属性还可以通过 HtmlHelper 的同名静态属性进行设置。值得一提的是，ASP.NET MVC 允许我们针对某个 View 开启或者关闭对客户端验证和 UnobtrusiveJavaScript 的支持，这可以通过 HtmlHelper 的实例方法 EnableClientValidation 和 EnableUnobtrusiveJavaScript 来实现。

```

public class HtmlHelper
{
    //其他成员
    public void EnableClientValidation();
    public void EnableClientValidation(bool enabled);
    public void EnableUnobtrusiveJavaScript();
    public void EnableUnobtrusiveJavaScript(bool enabled);
}

```

```

public static bool ClientValidationEnabled { get; set; }
public static bool UnobtrusiveJavaScriptEnabled { get; set; }
}

```

接口 `IView` 的 `Render` 方法的第二个参数是一个 `TextWriter` 对象。`View` 内容的输出可以通过针对 `TextWriter` 对象的写操作来实现，因为在 `View` 引擎在调用 `Render` 方法的时候，作为该参数的是当前 `HttpResponse` 的 `Output` 属性表示的 `TextWriter`。

8.2.2 ViewEngine

`View` 引擎的核心是 `ViewEngine` 对象，它实现了 `System.Web.Mvc.IViewEngine` 接口。如下面的代码片段所示，`IViewEngine` 定义了两个 `FindView` 和 `FindPartialView` 方法，它们根据指定的 `ControllerContext`、`View` 名称和布局文件名称获取对应的 `View` 和 `Partial View`。这两个方法均具有一个表示是否启用缓存的参数 `useCache`。另一个方法 `ReleaseView` 用于释放 `View` 对象。

```

public interface IViewEngine
{
    ViewEngineResult FindPartialView(ControllerContext controllerContext,
        string partialViewName, bool useCache);
    ViewEngineResult FindView(ControllerContext controllerContext,
        string viewName, string masterName, bool useCache);
    void ReleaseView(ControllerContext controllerContext, IView view);
}

```

`FindView` 和 `FindPartialView` 方法的返回类型并非 `IView`，而是一个用于封装 `View` 的类型 `System.Web.Mvc.ViewEngineResult`。如下面的代码片段所示，`ViewEngineResult` 的只读属性 `View` 和 `ViewEngine` 属性表示找到的 `View` 对象和作为调用者的 `ViewEngine` 对象。在成功获取到对应 `View` 的情况下这两个属性会通过构造函数进行初始化。如果没有找到相应的 `View`，则将表示搜寻位置的字符串列表传入另一个构造函数来创建返回的 `ViewEngineResult`，只读属性 `SearchedLocations` 表示的就是这么一个搜寻位置列表。

```

public class ViewEngineResult
{
    public ViewEngineResult(IEnumerable<string> searchedLocations);
    public ViewEngineResult(IView view, IViewEngine viewEngine);

    public IEnumerable<string> SearchedLocations { get; }
    public IView View { get; }
    public IViewEngine ViewEngine { get; }
}

```

如果返回的 `ViewEngineResult` 包含一个具体的 `View`，那么这个 `View` 将会最终被呈现出来。反之，如果 `ViewEngineResult` 仅仅包含一个通过 `SearchedLocations` 属性表示的搜索位置列表，那么最终呈现出来的就是如图 8-5 所示的包含该列表的错误页面。

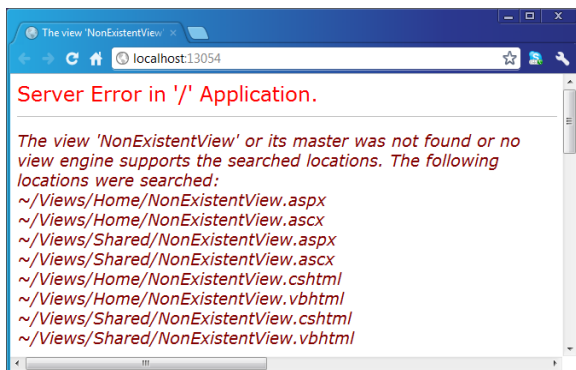


图 8-5 包含搜索位置列表的错误页面

我们可以通过一个简单的实例来验证这一点。在一个 ASP.NET MVC 应用中定义了如下一个 HomeController, 在默认的动作方法 Index 中, 我们通过 System.Web.Mvc.ViewEngines 的静态只读属性 Engines 得到一个全局 ViewEngine 列表, 并调用其 FindView 方法试图去获取一个根本不存在的 View (“NonExistentView”), 最后将得到的 ViewEngineResult 对象的 SearchedLocations 属性表示的搜寻位置列表呈现出来。

```
public class HomeController : Controller
{
    public void Index()
    {
        ViewEngineResult result = ViewEngines.Engines.FindView(
            ControllerContext, "NonExistentView", null);
        foreach (string location in result.SearchedLocations)
        {
            Response.Write(location + "<br/>");
        }
    }
}
```

运行程序后, 表示在获取目标 View 过程中采用的搜寻位置列表会以如图 8-6 所示的效果呈现出来, 这个列表的内容与图 8-5 是完全一致的。

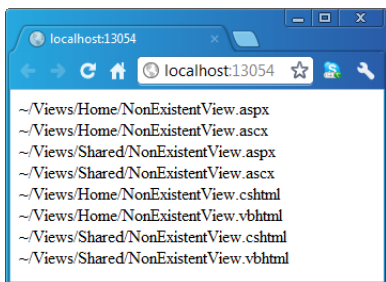


图 8-6 在获取目标 View 未果情况下采用的搜索位置列表

上面实例演示涉及到了一个重要的静态类型 ViewEngines, 它通过如下定义的只读属性 Engines 维护一个全局 ViewEngine 列表。从给出的定义可以看出, 两个原生的 ViewEngine

在初始化的时候就被添加到了该列表中，它们的类型就是分别代表 Web Form 和 Razor 引擎的 `System.Web.Mvc.WebFormViewEngine` 和 `System.Web.Mvc.RazorViewEngine`。如果我们创建了一个自定义 View 引擎，相应的 ViewEngine 也可以通过 ViewEngines 进行注册。

```
public static class ViewEngines
{
    private static readonly ViewEngineCollection _engines =
        new ViewEngineCollection {
            new WebFormViewEngine(), new RazorViewEngine() };

    public static ViewEngineCollection Engines
    {
        get { return _engines; }
    }
}

public class ViewEngineCollection : Collection<IViewEngine>
{
    //其他成员
    public virtual ViewEngineResult FindPartialView(
        ControllerContext controllerContext, string partialViewName);
    public virtual ViewEngineResult FindView(
        ControllerContext controllerContext, string viewName, string masterName);
}
```

`ViewEngines` 的静态只读属性 `Engines` 的类型是 `System.Web.Mvc.ViewEngineCollection`，它是一个元素类型为 `IViewEngine` 的集合。`ViewEngineCollection` 同样定义了 `FindView/FindPartialView` 方法用于获取指定名称的 View 和分部 View，这两个方法利用包含的 `ViewEngine` 对象进行 View 和 Partial View 的获取。由于 `WebFormViewEngine` 排在 `RazorViewEngine` 之前，所以前者会被优先使用，这可以从图 8-5/8-6 所示的搜寻位置列表看出来（先搜索.aspx 和.ascx，再搜索.cshtml 和.vbhtml）。

对于 `ViewEngineCollection` 的 `FindView/FindPartialView` 方法来说，不知道读者是否注意到了它们没有一个表示是否采用缓存的 `useCache` 参数。实际上当这两个方法被调用的时候，ASP.NET MVC 会先采用缓存的方式调用相应的 `ViewEngine`，如果返回值为 `Null`，再以不采用缓存的方式再次调用它们。

8.2.3 ViewResult 的执行

View 引擎对 View 的获取以及对 View 的呈现最初是通过 `ViewResult` 触发的，那么两者是如何衔接的呢？在正式讨论这个问题之前不妨先来看看 `ViewResult` 的定义。如下面的代码片段所示，表示 `ViewResult` 的类型 `System.Web.Mvc.ViewResult` 是抽象类 `System.Web.Mvc.ViewResultBase` 的子类。

```
public class ViewResult : ViewResultBase
{
    protected override ViewEngineResult FindView(ControllerContext context);
    public string MasterName { get; set; }
}
```

```

public abstract class ViewResultBase : ActionResult
{
    public override void ExecuteResult(ControllerContext context);
    protected abstract ViewEngineResult FindView(ControllerContext context);

    public object Model { get; }
    public TempDataDictionary TempData { get; set; }
    [Dynamic]
    public object ViewBag { [return: Dynamic] get; }
    public ViewDataDictionary ViewData { get; set; }
    public string ViewName { get; set; }
    public ViewEngineCollection ViewEngineCollection { get; set; }
    public IView View { get; set; }
}

```

`ViewResultBase` 的只读属性 `Model` 表示作为 `View` 的 `Model` 对象，三个表示数据状态的属性（`ViewData`、`ViewBag` 和 `TempData`）来源于 `Controller` 的同名属性。`View` 和 `ViewName` 属性则代表具体的 `View` 对象和 `View` 的名称。`ViewEngineCollection` 属性值默认来源于 `ViewEngines` 的静态属性 `Engines` 代表的全局 `ViewEngine` 列表。

`ViewResultBase` 用于获取具体 `View` 的抽象方法 `FindView` 在 `ViewResult` 中被实现，后者提供了额外的属性 `MasterName` 表示布局文件名称。`FindView` 方法在内部会直接调用 `ViewEngineCollection` 属性的 `FindView` 方法，如果返回的 `ViewEngineResult` 包含一个具体的 `View`（`View` 属性不为空），则直接将它返回，否则抛出一个 `InvalidOperationException` 异常，并将通过 `ViewEngineResult` 的 `SearchedLocations` 属性表示的搜寻位置列表格式化成一个字符串作为该异常的消息，所以图 8-5 所示的搜寻位置列表实际上是抛出的 `InvalidOperationException` 异常的消息。

ASP.NET MVC 的 `View` 引擎涉及到的相关的类型/接口以及它们之间的关系可以通过如图 8-7 所示的 UML 来表示。`ViewResult` 通过静态类型 `ViewEngines` 利用 `View` 引擎激活对应的 `View` 对象并最终将 `View` 的内容呈现出来。

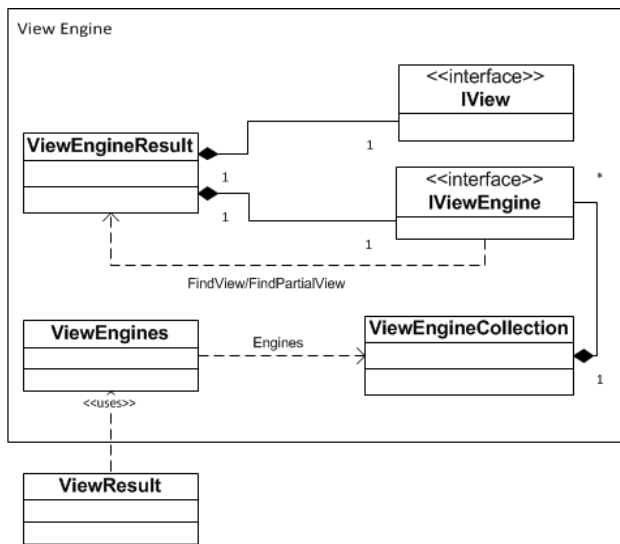


图 8-7 View 引擎与 `ViewResult`

与除 EmptyResult 以外的所有 ActionResult 类型一样, 抽象类 Controller 中提供了相应的方法辅助创建 ViewResult。如下面的代码片段所示, Controller 具有如下一系列 View 方法重载帮助我们根据指定的 View 名称、View 对象、布局文件名称和 Model 对象创建相应的 ViewResult。

```
public abstract class Controller : ControllerBase, ...
{
    //其他成员
    protected ViewResult View();
    protected ViewResult View(object model);
    protected ViewResult View(string viewName);
    protected ViewResult View(IView view);
    protected ViewResult View(string viewName, object model);
    protected ViewResult View(string viewName, string masterName);
    protected virtual ViewResult View(IView view, object model);
    protected virtual ViewResult View(string viewName,
        string masterName, object model);
}
```

ViewResult 与 View 引擎的交互体现在实现的 ExecuteResult 方法上。如下面的代码片段所示, 如果 View 属性为 Null, ASP.NET MVC 会调用 FindView 方法根据指定的 View 名称 (如果没有执行则采用当前的 Action 名称作为 View 名称) 得到一个 ViewEngineResult 对象, 并将其 View 属性作为自身的 View。接下来一个 ViewContext 被创建出来, 它和当前 HttpResponseMessage 的 Output 属性代表的 TextWriter 对象被作为参数调用 View 对象的 Render 方法实现对 View 的呈现。View 呈现完成之后, ViewEngineResult 对应的 ViewEngine 被提取出来, 并调用其 Release 方法释放 View 对象。

```
public abstract class ViewResultBase : ActionResult
{
    //其他成员
    public override void ExecuteResult(ControllerContext context)
    {
        //其他操作
        if (string.IsNullOrEmpty(this.ViewName))
        {
            this.ViewName = context.RouteData.GetRequiredString("action");
        }
        ViewEngineResult result = null;
        if (this.View == null)
        {
            result = this.FindView(context);
            this.View = result.View;
        }
        TextWriter output = context.HttpContext.Response.Output;
        ViewContext viewContext = new ViewContext(context, this.View,
            this.ViewData, this.TempData, output);
        this.View.Render(viewContext, output);
        if (result != null)
        {
            result.ViewEngine.ReleaseView(context, this.View);
        }
    }
}
```

```

    }
}

```

ViewResult 为我们提供了一种与 View 引擎交互的捷径,其实我们在进行 View 的获取和呈现的时候完全可以抛开 ViewResult, 直接利用 View 引擎来完成。如下所示的两种 Action 方法的定义是完全等效的。

```

//Action 方法直接返回 ViewResult
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}

//Action 方法直接调用 View 引擎
public class HomeController : Controller
{
    public void Index()
    {
        string viewName = ControllerContext.RouteData
            .GetRequiredString("action");
        ViewEngineResult result = ViewEngines.Engines.FindView(ControllerContext,
            viewName, null);
        if (null == result.View)
        {
            throw new InvalidOperationException(FormatErrorMessage(
                viewName, result.SearchedLocations));
        }
        try
        {
            ViewContext viewContext = new ViewContext(ControllerContext,
                result.View, this.ViewData, this.TempData, Response.Output);
            result.View.Render(viewContext, viewContext.Writer);
        }
        finally
        {
            result.ViewEngine.ReleaseView(ControllerContext, result.View);
        }
    }

    private string FormatErrorMessage(string viewName,
        IEnumerable<string>searchedLocations)
    {
        string format = "The view '{0}' or its master was not found or no view
engine supports the searched locations. The following locations were
searched:{1}";
        StringBuilder builder = new StringBuilder();
        foreach (string str in searchedLocations)
        {
            builder.AppendLine();
            builder.Append(str);
        }
    }
}

```

```

    }
    return string.Format(CultureInfo.CurrentCulture, format, viewName,
        builder);
}
}

```

前面仅仅介绍了 ViewResult 利用 View 引擎进行 View 的获取和呈现，其实当我们调用 HtmlHelper 的扩展方法 Partial 进行 Partial View 呈现时，内部调用 View 引擎的方式与之类似。

实例演示：创建自定义 View (S805)

为了让读者对 View 引擎及其 View 呈现机制具有一个深刻的认识，我们自定义一个简单的用于呈现静态 HTML 的 StaticFileViewEngine。在一个 ASP.NET MVC 应用中定义了如下一个实现了 IView 接口的类型 StaticFileView，它表示一个用于呈现指定文件内容的自定义 View，在实现的 Render 方法中指定文件的内容被读取出来写入作为参数的 TextWriter 对象中。

```

public class StaticFileView:IView
{
    public string FileName { get; private set; }
    public StaticFileView(string fileName)
    {
        this.FileName = fileName;
    }
    public void Render(ViewContext viewContext, TextWriter writer)
    {
        byte[] buffer;
        using (FileStream fs = new FileStream(this.FileName, FileMode.Open))
        {
            buffer = new byte[fs.Length];
            fs.Read(buffer, 0, buffer.Length);
        }
        writer.Write(Encoding.UTF8.GetString(buffer));
    }
}

```

由于 StaticFileView 中定义的内容完全是静态的，所以缓存显得很有必要。我们只需要基于 Controller 和 View 名称来实施缓存，为此定义了如下一个作为 Key 的数据类型 ViewEngineResultCacheKey。

```

internal class ViewEngineResultCacheKey
{
    public string ControllerName { get; private set; }
    public string ViewName { get; private set; }

    public ViewEngineResultCacheKey(string controllerName, string viewName)
    {
        this.ControllerName = controllerName ?? string.Empty;
        this.ViewName = viewName ?? string.Empty;
    }
    public override int GetHashCode()
    {
        return this.ControllerName.ToLower().GetHashCode()

```

```

        ^ this.ViewName.ToLower().GetHashCode();
    }

    public override bool Equals(object obj)
    {
        ViewEngineResultCacheKey key = obj as ViewEngineResultCacheKey;
        if (null == key)
        {
            return false;
        }
        return key.GetHashCode() == this.GetHashCode();
    }
}

```

具有如下定义的 `StaticFileViewEngine` 代表 `StaticFileView` 对应的 `ViewEngine`。我们通过一个字典类型的字段 `viewEngineResults` 来全局缓存创建的 `ViewEngineResult`，而 `View` 的获取实现在 `InternalFindView` 方法中。通过 `StaticFileView` 表示的 `View` 定义在一个以 `View` 名称作为文件名的文本文件中，该文件的扩展名为 `.shtml`（Static HTML）。

```

public class StaticFileViewEngine : IViewEngine
{
    private Dictionary<ViewEngineResultCacheKey, ViewEngineResult>
        viewEngineResults =
            new Dictionary<ViewEngineResultCacheKey, ViewEngineResult>();
    private object syncHelper = new object();
    public ViewEngineResult FindPartialView(ControllerContext controllerContext,
        string partialViewName, bool useCache)
    {
        return this.FindView(controllerContext, partialViewName, null, useCache);
    }

    public ViewEngineResult FindView(ControllerContext controllerContext,
        string viewName, string masterName, bool useCache)
    {
        string controllerName =
            controllerContext.RouteData.GetRequiredString("controller");
        ViewEngineResultCacheKey key =
            new ViewEngineResultCacheKey(controllerName, viewName);
        ViewEngineResult result;
        if (!useCache)
        {
            result = InternalFindView(controllerContext, viewName,
                controllerName);
            viewEngineResults[key] = result;
            return result;
        }
        if (viewEngineResults.TryGetValue(key, out result))
        {
            return result;
        }
        lock (syncHelper)
        {
            if (viewEngineResults.TryGetValue(key, out result))
            {

```

```

        return result;
    }

    result = InternalFindView(controllerContext, viewName,
        controllerName);
    viewEngineResults[key] = result;
    return result;
}

private ViewEngineResult InternalFindView(ControllerContext
    controllerContext, string viewName, string controllerName)
{
    string[] searchLocations = new string[]
    {
        string.Format("~/views/{0}/{1}.shtml", controllerName, viewName),
        string.Format("~/views/Shared/{0}.shtml", viewName)
    };

    string fileName =
        controllerContext.HttpContext.Request.MapPath(searchLocations[0]);
    if (File.Exists(fileName))
    {
        return new ViewEngineResult(new StaticFileView(fileName), this);
    }
    fileName = string.Format(@"\views\Shared\{0}.shtml", viewName);
    if (File.Exists(fileName))
    {
        return new ViewEngineResult(new StaticFileView(fileName), this);
    }
    return new ViewEngineResult(searchLocations);
}

public void ReleaseView(ControllerContext controllerContext, IView view)
{ }
}

```

在 `InternalFindView` 方法中，我们先在 “~/Views/{ControllerName}/” 目录下寻找 View 文件，如果不存在则在目录 “~/Views/Shared/” 中寻找。如果对应 View 文件被找到，则以此创建一个 `StaticFileView` 对象，并将其封装在返回的 `ViewEngineResult` 对象中。如果目标 View 文件找不到，则根据基于这两个目录的搜寻地址列表创建并返回对应的 `ViewEngineResult`。

现在我们在 `Global.asax` 中通过如下的代码对自定义的 `StaticFileViewEngine` 进行注册，我们将创建的 `StaticFileViewEngine` 作为第一个使用的 ViewEngine。

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        ViewEngines.Engines.Insert(0, new StaticFileViewEngine());
    }
}

```


接下来定义了如下一个简单的 HomeController，Action 方法 ShowNonExistentView 中通过调用 View 方法呈现一个不存在的 View（View 名称为“NonExistentView”），而 ShowStaticFileView 方法则将对应的 StaticFileView 呈现出来。

```
public class HomeController : Controller
{
    public ActionResult ShowNonExistentView()
    {
        return View("NonExistentView");
    }

    public ActionResult ShowStaticFileView()
    {
        return View();
    }
}
```

我们为 Action 方法 ShowStaticFileView 定义了一个具有如下所示 HTML 的 View 文件 ShowStaticFileView.shtml（扩展名不是.cshtml，而是.shtml），并将其保存在“~/Views/Home”目录下。

```
<html>
  <head>
    <title>Static File View</title>
  </head>
  <body>
    这是一个自定义的 StaticFileView!
  </body>
</html>
```

运行程序并在浏览器中输入相应的地址访问 Action 方法 ShowNonExistentView，会得到如图 8-8 所示的输出结果。图中列出的 View 搜寻位置列表中的前两项正是我们自定义的 StaticFileEngine 寻找对应.shtml 文件的两个路径。

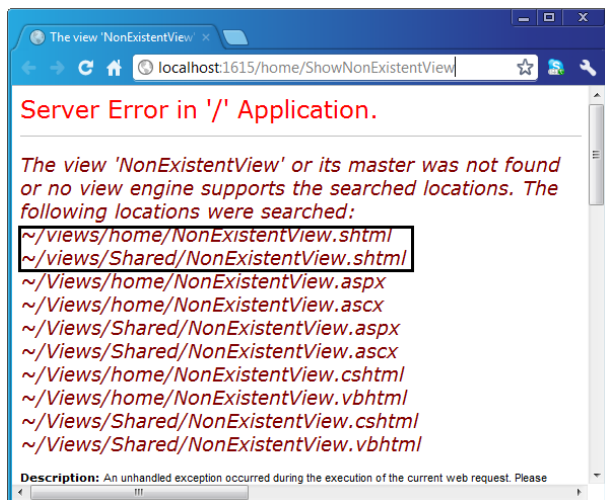


图 8-8 自定义 ViewEngine 在获取 View 失败情况下对搜寻地址列表的显示

如果我们改变浏览器的地址来访问另一个 Action 方法 ShowStaticFileView，会呈现出如图 8-9 所示的输出结果，不难看出呈现出来的正是定义在 ShowStaticFileView.shtml 中的 HTML。

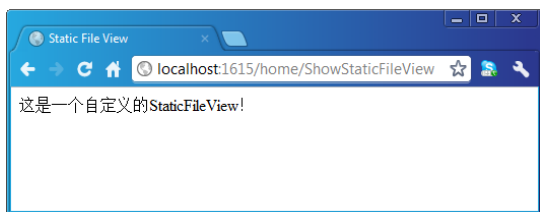


图 8-9 StaticFileView 针对静态文件内容的呈现

8.3 Razor 引擎

Web Form 引擎(或者 ASPX 引擎)和 Razor 引擎是 ASP.NET MVC 原生支持的两种 View 引擎，但从编程便捷性来看，后者无疑是更好的选择，所以我们选择对 Razor 引擎的运行机制做一个详细介绍。View 引擎实现了 View 对象的创建与呈现，对于 Razor 引擎来说，具体的 View 的内容被定义在相应的.cshtml 或者.vbhtml 文件中，表面来看 View 引擎帮助我们根据指定的 View 名称按照预定义的目录列表来搜寻相应的 View 文件，但是不要忘了 ASP.NET MVC 不能“解释”View 文件，View 文件必须经过编译。

8.3.1 View 的编译原理

通过.cshtml 或者.vbhtml 文件定义的 View 能够被执行，必须先被编译成存在于某个程序集的类型，ASP.NET MVC 采用动态编译的方式对 View 文件实施编译。当我们在对 ASP.NET MVC 进行部署的时候，需要对.cshtml 或者.vbhtml 文件进行打包。针对某个 View 的第 1 次访问会触发针对它的编译，一个 View 对应着一个类型。

我们可以对.cshtml 或者.vbhtml 进行修改，View 文件修改后的第 1 次访问将会导致 View 的再一次编译。和 ASP.NET 传统的编译方式一样，针对 View 的编译默认是基于目录的，也就是说同一个目录下的多个 View 文件被编译到同一个程序集中。

实例演示：探测基于目录的 View 编译机制（S806）

为了让读者对 ASP.NET MVC 中 View 文件的编译机制有一个深刻的认识，我们通过一个简单的实例来确定 View 文件最终都被编译成什么类型，所在的程序集又是哪一个。我们在一个 ASP.NET MVC 应用中为 HtmlHelper 定义了如下一个扩展方法 ListViewAssemblies，该方法用于获取当前被加载的包含 View 类型的程序集（程序集名称以“App_Web_”为前缀）。

```
public static class HtmlHelperExtensions
{
```

```

public static MvcHtmlString ListViewAssemblies(this HtmlHelper helper)
{
    TagBuilder ul = new TagBuilder("ul");
    foreach(var assembly in AppDomain.CurrentDomain.GetAssemblies()
        .Where(a=>a.FullName.StartsWith("App_Web_")))
    {
        TagBuilder li = new TagBuilder("li");
        li.InnerHtml = assembly.FullName;
        ul.InnerHtml+= li.ToString();
    }
    return new MvcHtmlString(ul.ToString());
}

```

然后我们定义了如下两个 Controller 类型 (FooController 和 BarController)，它们之中各自定义了两个 Action 方法 Action1 和 Action2。

```

public class FooController : Controller
{
    public ActionResult Action1()
    {
        return View();
    }
    public ActionResult Action2()
    {
        return View();
    }
}

public class BarController : Controller
{
    public ActionResult Action1()
    {
        return View();
    }
    public ActionResult Action2()
    {
        return View();
    }
}

```

接下来我们为定义在 FooController 和 BarController 的四个 Action 创建对应的 View (对应文件路径为：“~/Views/Foo/Action1.cshtml”、“~/Views/Foo/Action2.cshtml”、“~/Views/Bar/Action1.cshtml”和“~/Views/Bar/Action2.cshtml”)。它们具有如下相同的定义，我们在 View 中显示自身的类型和当前加载的基于 View 的程序集。

```

<div>当前 View 类型:@this.GetType().AssemblyQualifiedName</div>
<div>当前加载的 View 程序集:</div>
@Html.ListViewAssemblies()

```

现在运行我们的程序并在浏览器中通过输入相应的地址“依次”(“Foo/Action1”、“Foo/Action2”、“Bar/Action1”和“Bar/Action2”)访问定义在 FooController 和 BarController 的四个 Action，四次访问得到的输出结果如图 8-10 所示。

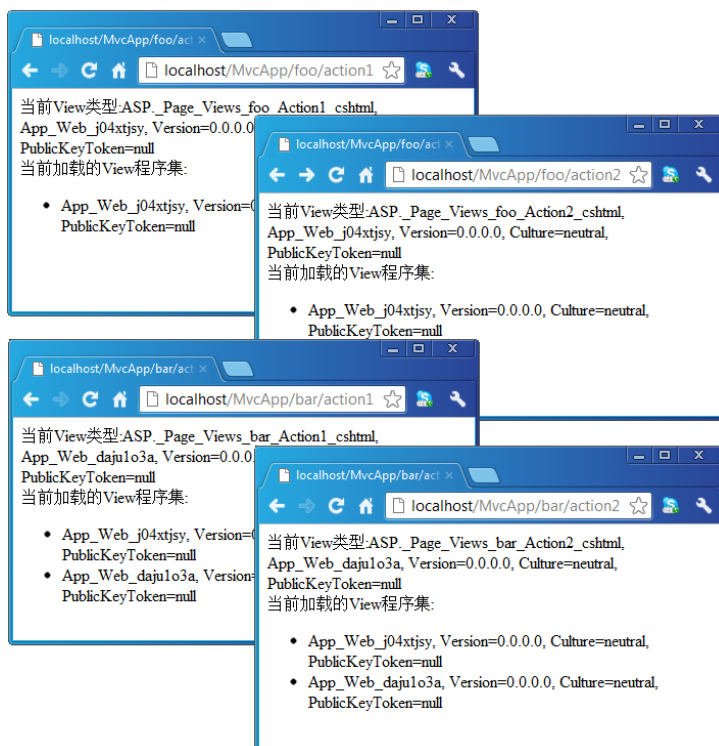


图 8-10 当前 View 的类型和加载的 View 程序集

图 8-10 所示的输出结果至少可以反映三个问题。

- ASP.NET MVC 对 View 文件进行动态编译生成的类型名称基于 View 文件的虚拟路径（比如文件路径为“~/Views/Foo/Action1.cshtml”的 View 对应的类型为“ASP_Page_VIEWS_foo_Action1_cshtml”）。
- ASP.NET MVC 是按照目录进行编译的（“~/Views/Foo/”下的两个 View 文件最终都被编译到程序集“App_Web_j04xtjsy”中）。
- 程序集是按需加载的。第一次访问“~/View/Foo/”目录下的 View 并不会加载针对“~/View/Bar/”目录的程序集（实际上此时该程序集尚未生成）。

我们可以调用 BuildManager 类型的静态方法 GetCompiledType 和 GetCompiledAssembly（如下面的代码片段所示）根据 View 文件的虚拟路径分别得到编译后的类型和程序集。

```
public sealed class BuildManager
{
    //其他成员
    public static Type      GetCompiledType(string virtualPath);
    public static Assembly  GetCompiledAssembly(string virtualPath);
}
```

在现有演示实例的基础上我们创建了如下一个 HomeController，默认的 Action 方法 Index

中通过调用 `BuildManager` 的静态方法 `GetCompiledType` 得到并呈现出四个 View 文件对应的类型名称。

```
public class HomeController : Controller
{
    public void Index()
    {
        Response.Write(BuildManager.GetCompiledType(
            "~/Views/Foo/Action1.cshtml") + "<br/>");
        Response.Write(BuildManager.GetCompiledType(
            "~/Views/Foo/Action2.cshtml") + "<br/>");
        Response.Write(BuildManager.GetCompiledType(
            "~/Views/Bar/Action1.cshtml") + "<br/>");
        Response.Write(BuildManager.GetCompiledType(
            "~/Views/Bar/Action2.cshtml") + "<br/>");
    }
}
```

直接运行程序后会在浏览器中得到代表四个 View 文件编译类型名称的字符串，具体显示效果如图 8-11 所示。与图 8-10 显示的 View 类型名称相比较，我们会发现它们是一致的。

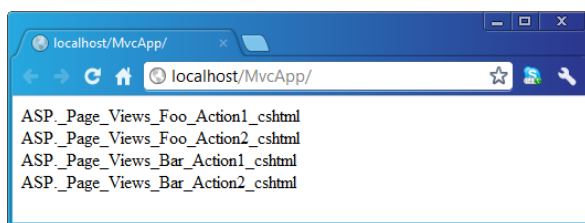


图 8-11 View 文件编译后的类型名称

编译 View 文件会生成怎样的类型

前面我们简单地介绍了 ASP.NET MVC 以目录为单位的动态 View 编译，有人可能会问一个问题：编译生成的程序集存放在哪里？在默认情况下，View 文件被动态编译后生成的程序集被临时存放在 ASP.NET 的临时目录 “%WinDir%\Microsoft.NET\Framework\{Version No}\Temporary ASP.NET Files\” 下，不过我们可以通过如下所示的配置节 `<system.web>` / `<compilation>` 的 `tempDirectory` 属性来改变动态编译的临时目录。如果我们改变了这个临时目录，需要确保工作进程运行账号具有访问该目录的权限。

```
<configuration>
  <system.web>
    <compilation tempDirectory="c:\Temporary ASP.NET Files\" .../>
  </system.web>
</configuration>
```

一个寄宿于 IIS 的 Web 应用会在上述的临时目录下创建一个与 Web 应用同名的子目录，所以我们可以很容易地找到应用对应的编译目录。但是对于将 Visual Studio Development Server 作为宿主的 Web 应用都会编译到名称为 `Root` 的子目录下。如果这样的应用太多，我们往往不太容易准确地找到基于某个应用的编译目录。有时候可以根据目录最后的修改时间来找到它，但是笔者个人倾向于直接删除整个 `Root` 目录，然后运行我们的程序后会重新生成一个

只包含该应用编译目录的 Root 目录。

对于上面演示的实例，笔者将 Web 应用寄宿于 IIS 下并且命名为 MvcApp，在本机的目录“C:\Windows\Microsoft.NET\Framework\v4.0.30319\Temporary ASP.NET Files\mvcapp\c4ea0afa\83bd407”下可以找到动态编译的生成的文件。如图 8-12 所示，两个 View 目录（“~/Views/Foo”和“~/Views/Bar”）编译生成的程序集就在这个目录下面。

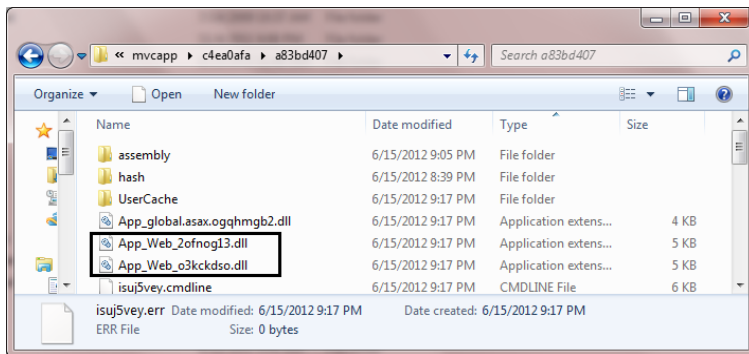


图 8-12 ASP.NET 临时目录中编译生成的程序集

读者一定很好奇一个 View 文件通过动态编译最终会生成一个怎样的类型？对应前面演示的实例我们已经知道了四个 View 文件编译生成的类型名称和所在的程序集，我们只需要通过 Reflector 打开对应的程序集就能得到 View 文件编译类型的定义。如下所示的是 View 文件“~/Views/Foo/Action.cshtml”编译后生成的 ASP_Page_VIEWS_Foo_Action1_cshtml 类型的定义。

```
[Dynamic(new bool[] { false, true })]
public class _Page_VIEWS_Foo_Action1_cshtml : WebViewPage<object>
{
    public override void Execute()
    {
        this.WriteLiteral("<div>当前 View 类型: </div>\r\n<div>");
        this.Write(base.GetType().AssemblyQualifiedName);
        this.WriteLiteral("</div><br/>\r\n<div>当前加载的 View 程序集: </div>\r\n");
        this.Write(base.Html.ListViewAssemblies());
    }

    protected global_asax ApplicationInstance
    {
        get
        {
            return (global_asax) this.Context.ApplicationInstance;
        }
    }
}
```

8.3.2 WebViewPage 与 WebViewPage<TModel>

从上面的代码可以看出 View 文件编译生成的类型是 System.Web.Mvc.WebViewPage

<TModel>的子类，泛型参数 TModel 代表 View 的 Model 类型。WebViewPage<TModel>是 System.Web.Mvc.WebViewPage 的子类，不论强类型还是弱类型 View 文件，最终编译生成的都是 WebViewPage<TModel>子类，弱类型 View 类型继承自 WebViewPage<object>（比如上面的 _Page_Views_Foo_Action1_cshtml）。

WebViewPage 定义和继承了很多成员，由于 View 文件编译的目标类型是 WebViewPage 的子类，意味着这些成员可以直接在定义 View 的时候使用，所以很有必要对它们有一个基本的了解。受篇幅所限，在这里我们只对一些常用的属性和方法作一个大概的介绍。

WebPageExecutingBase

WebViewPage 可以简单地视为一个 WebPage，它最终继承自具有如下定义的抽象类 System.Web.WebPages.WebPageExecutingBase。我们通过调用抽象方法 Execute 将页面内容呈现出来，View 文件编译生成的类型正是通过重写这个方法将自身定义的内容呈现出来的。具体来说，View 文件的内容大体可以分为静态和动态两个部分，静态内容被原样呈现，而动态内容通过执行一段代码来生成。静态内容和动态内容的输出分别通过调用 WriteLiteral 和 Write 方法来完成。除了这三个重要的抽象方法之外，WebViewPage 还具有一个表示虚拟路径的 VirtualPath 属性。

```
public abstract class WebPageExecutingBase
{
    //其他成员
    public abstract void Execute();
    public abstract void Write(object value);
    public abstract void WriteLiteral(object value);

    public virtual string VirtualPath { get; set; }
}
```

WebPageRenderingBase

另一个抽象类 System.Web.WebPages.WebPageRenderingBase 是 WebPageExecutingBase 的子类。我们知道一个 View 文件可以具有一个外部的布局文件（或者母版页），布局文件的虚拟路径通过抽象属性 Layout 表示，在定义 View 的时候对 Layout 文件的设置实际上就是对该属性赋值。View 自身的内容通过重写的 Execute 方法输出，整个页面（包括布局文件和 View 文件）的内容输出通过调用抽象方法 ExecutePageHierarchy 来完成。

```
public abstract class WebPageRenderingBase : WebPageExecutingBase, ...
{
    //其他成员
    public abstract void ExecutePageHierarchy();
    public abstract string Layout { get; set; }

    public virtual bool IsAjax { get; }
}
```

```

public virtual bool IsPost { get; }
public string      Culture { get; set; }
public string      UICulture { get; set; }

public virtual HttpRequestBase      Request { get; }
public virtual HttpResponseBase     Response { get; }
public virtual HttpServerUtilityBase Server { get; }
public virtual HttpSessionStateBase Session { get; }
public virtual IPrincipal           User { get; }
public ProfileBase                  Profile { get; }
public virtual Cache                 Cache { get; }
}

```

`WebPageRenderingBase` 还定义了一些辅助的属性帮助我们判断当前的请求是 Ajax 请求（`IsAjax`）还是 Post 请求（`IsPost`）。通过属性 `Culture`/`UICulture` 可以获取和设置当前线程的语言文化，通过其他相应的属性还可以获取基于当前 `HttpContext` 的一些属性（`Request`、`Response`、`Server`、`Session`、`Profile` 和 `Cache` 等）。

WebPageBase

`System.Web.WebPages.WebPageBase` 是 `WebPageRenderingBase` 的子类，也是 `WebViewPage` 的直接基类。除了实现定义在 `WebPageRenderingBase` 中的抽象方法 `ExecutePageHierarchy` 方法之外，`WebPageBase` 定义了额外两个 `ExecutePageHierarchy` 方法重载。在调用这两个方法的时候，我们需要将执行的上下文信息封装在类型为 `System.Web.WebPages.WebPageContext` 的参数 `pageContext` 中，而另一个参数 `startPage` 表示定义在 `_ViewStart.cshtml/_ViewStart.vbhtml` 中的开始页面。最终呈现出来的页面的内容来源于三个部分：布局文件、开始页面和 `View` 本身的内容。完整页面内容的呈现是通过调用这两个 `ExecutePageHierarchy` 方法来完成的。

```

public abstract class WebPageBase : WebPageRenderingBase
{
    //其他成员
    public override void ExecutePageHierarchy();
    public void ExecutePageHierarchy(WebPageContext pageContext,
        TextWriter writer);
    public void ExecutePageHierarchy(WebPageContext pageContext,
        TextWriter writer, WebPageRenderingBase startPage);

    public override void Write(object value);
    public override void WriteLiteral(object value);
    public override string Layout { get; set; }

    public HelperResult RenderSection(string name);
    public HelperResult RenderSection(string name, bool required);
    public HelperResult RenderBody();

    public bool IsSectionDefined(string name);
    public void DefineSection(string name, SectionWriter action);
}

```

`WebPageBase` 实现了定义在 `WebPageExecutingBase` 上的抽象方法 `Write`/`WriteLiteral` 和抽

象属性 `Layout`。除此之外, `WebPageBase` 还定义了一些帮助我们在布局文件上输出 `Section` 的方法, 其中 `RenderSection` 用于输出指定名称的 `Section`, 参数 `required` 表示输出的 `Section` 是否是必需的。如果该参数值为 `True`, 在相应的 `Section` 尚未定义的情况下会直接抛出 `HttpException` 异常。`RenderBody` 用于输出主体部分 (名称为 “Body” 的 `Section`), 而 `IsSectionDefined` 用于判断指定名称的 `Section` 是否被定义。方法 `DefineSection` 用于定义相应的 `Section`, 在 `View` 文件中定义 `Section` 的 `@section sectionName{...}` 语句最终都转换为针对该方法的调用。

比如说我们通过如下一个布局文件将页面分为 `Header`、`Body` 和 `Footer` 三个 `Section`, 如果 `View` 中没有显式定义 `Header` 和 `Footer`, 则采用默认定义的内容进行填充。使用该布局文件的 `View` 定义了 `Body` 的内容 (`View` 中没有显式指定 `Section` 名称的内容被默认作为 `Body` 的内容), 并且定义了 `Header` 这个 `Section`。

```
//布局文件定义
<!DOCTYPE html>
<html>
  <head>
    <title>@ViewBag.Title</title>
  </head>
  <body>
    @{if(IsSectionDefined("Header"))
      {@RenderSection("Header");}
      else{<h2>@ViewBag.Title</h2>}}
    }
    @RenderBody()
    @{if (IsSectionDefined("Footer"))
      {@RenderSection("Footer");}
      else{<hr/><p>©2012 Artech. All rights reserved.</p>}}
    }
  </body>
</html>

//View 的定义
<div>
  <p>dummy text...dummy text...dummy text...dummy text...dummy text...
    dummy text...</p>
  <p>dummy text...dummy text...dummy text...dummy text...dummy text...
    dummy text...</p>
</div>

@section Header
{
  <h2>Article Title</h2>
}
```

上面定义的 `View` 最终呈现在浏览器中的效果如图 8-13 所示, 我们可以看到 `Header`、`Body` 和 `Footer` 的内容得到了合理的显示, 其中 `Header` 和 `Footer` 的内容分别来源于 `View` 和 布局文件。



图 8-13 采用了布局文件的 View 的呈现效果

WebViewPage

View 文件最终编译生成的 `WebViewPage<TModel>` 是 `System.Web.Mvc.WebViewPage` 的子类，而后者继承自 `WebPageBase`。我们在定义 View 的时候使用的三个帮助对象（`HtmlHelper`、`UrlHelper` 和 `AjaxHelper`）所对应的属性就定义在这里，它们通过调用 `InitHelpers` 方法进行初始化。

```
public abstract class WebViewPage : WebPageBase, IViewDataContainer,
    IViewStartPageChild
{
    //其他成员
    public HtmlHelper<object> Html { get; set; }
    public UrlHelper          Url { get; set; }
    public AjaxHelper<object> Ajax { get; set; }
    public virtual void        InitHelpers();

    public ViewContext          ViewContext { get; set; }
    public ViewDataDictionary ViewData { get; set; }
    [Dynamic]
    public object               ViewBag { get; }
    public TempDataDictionary TempData { get; }
    public object               Model { get; }

    protected virtual void SetViewData(ViewDataDictionary viewData);
}
```

我们在定义 View 的时候可以通过只读属性 `Model`、`ViewData`、`ViewBag` 和 `TempData` 得到在 Controller 中设置的状态数据，也可以通过调用 `SetViewData` 设置 `ViewData`，`ViewContext` 表示 View 当前上下文。

WebViewPage<TModel>

虽然 View 具有强弱类型之分，但是 View 文件生成的类型都是“强类型的”，它们都是 `WebViewPage<TModel>` 的子类。对于弱类型 View 来说，生成的类型继承自 `WebViewPage<object>`。如下面的代码片段所示，`WebViewPage<TModel>` 的属性 `Ajax`、`Html`、

Model 和 ViewData 属性都是针对泛型参数 TModel 的，而初始化它们的 InitHelpers 和 SetViewData 方法被重写。

```
public abstract class WebViewPage<TModel> : WebViewPage
{
    public override void InitHelpers();
    protected override void SetViewData(ViewDataDictionary viewData);

    public AjaxHelper<TModel> Ajax { get; set; }
    public HtmlHelper<TModel> Html { get; set; }
    public TModel Model { get; }
    public ViewDataDictionary<TModel> ViewData { get; set; }
}
```

8.3.3 RazorView

Razor 引擎下的 View 通过类型 System.Web.Mvc.RazorView 表示，它与表示 Web Form 引擎 View 的类型 System.Web.Mvc.WebFormView 都是 System.Web.Mvc.BuildManagerCompiledView 的子类。

BuildManagerCompiledView

为了能够清楚地说明实现在 BuildManagerCompiledView 中的 View 激活与呈现机制，我们列出了 BuildManagerCompiledView 中与此相关的内部和受保护的成员。

```
public abstract class BuildManagerCompiledView : IView
{
    internal IViewPageActivator ViewPageActivator;

    protected BuildManagerCompiledView(ControllerContext controllerContext,
        string viewPath);
    protected BuildManagerCompiledView(ControllerContext controllerContext,
        string viewPath, IViewPageActivator viewPageActivator);
    internal BuildManagerCompiledView(ControllerContext controllerContext,
        string viewPath, IViewPageActivator viewPageActivator,
        IDependencyResolver dependencyResolver);

    public void Render(ViewContext viewContext, TextWriter writer);
    protected abstract void RenderView(ViewContext viewContext,
        TextWriter writer, object instance);

    internal IBuildManager BuildManager { get; set; }
    public string ViewPath { get; protected set; }
}
```

采用 Razor 引擎的 View 文件（.cshtml 或者.vbhtml）最终都会编译成一个 WebViewPage 类型，所以通过 RazorView/WebFormView 体现的 View 的呈现机制最终体现在对 WebViewPage 对象的激活上。根据上面介绍的 ASP.NET MVC 编译原理，可以利用 BuildManager 根据 View 文件的虚拟路径得到编译后的类型。从名称也可以看出来，BuildManagerCompiledView 内部就是利用了 BuildManager 根据指定的 View 文件虚拟路径完

成对 `WebViewPage` 对象激活的。

`BuildManagerCompiledView` 的属性 `ViewPath` 表示的就是 `View` 文件的虚拟路径, 该属性在构造函数中被初始化。`BuildManagerCompiledView` 具有三个构造函数, 对象本身的构造逻辑体现在内部构造函数上。如上面的代码片段所示, 除了将当前 `ControllerContext` 和 `View` 文件虚拟路径作为构造函数的参数之外, 该构造函数还具有额外两个参数, 其类型分别是 `System.Web.Mvc.IViewPageActivator` 和 `IDependencyResolver`。

```
public interface IViewPageActivator
{
    object Create(ControllerContext controllerContext, Type type);
}
```

上面的代码片段体现了接口 `IViewPageActivator` 的定义。顾名思义, 该接口旨在实现对 `WebViewPage` 对象的激活, 基于类型的对象激活机制实现在 `Create` 方法中。`BuildManagerCompiledView` 的构造函数中指定的 `ViewPageActivator` 被用于初始化内部字段 `ViewPageActivator`, 默认采用的是一个 `DefaultViewPageActivator` 对象。

`DefaultViewPageActivator` 是一个具有如下定义的内部类型, 可以看到它实际上依赖于一个 `DependencyResolver` 对象完成针对 `WebViewPage` 对象的激活。这个 `DependencyResolver` 对象可以通过构造函数进行显式设置, 而默认使用的 `DependencyResolver` 对象来源于 `DependencyResolver` 类型的静态属性 `Current`。

```
internal class DefaultViewPageActivator : IViewPageActivator
{
    private Func<IDependencyResolver> _resolverThunk;
    public DefaultViewPageActivator() : this(null)
    {}

    public DefaultViewPageActivator(IDependencyResolver resolver)
    {
        Func<IDependencyResolver> func = null;
        if (resolver == null)
        {
            this._resolverThunk = () => DependencyResolver.Current;
        }
        else
        {
            if (func == null)
            {
                func = () => resolver;
            }
            this._resolverThunk = func;
        }
    }

    public object Create(ControllerContext controllerContext, Type type)
    {
        return (this._resolverThunk().GetService(type)
            ?? Activator.CreateInstance(type));
    }
}
```

如果我们在构造 `BuildManagerCompiledView` 的时候没有指定具体的 `ViewPageActivator`，那么 ASP.NET MVC 会根据指定的 `DependencyResolver` 来创建默认的 `DefaultViewPageActivator`。如果我们只是根据 `ControllerContext` 和 `View` 文件虚拟路径来构建 `BuildManagerCompiledView`，最终用于激活 `WebPageView` 的实际上就是当前的 `DependencyResolver`。换句话说，我们可以通过注册自定义 `DependencyResolver` 的方法以 IoC 的方式来实现对 `WebPageView` 的激活，接下来我们会演示相关的实例。

`BuildManagerCompiledView` 对 View 的呈现机制其实很简单。它调用 `BuildManager` 的静态方法 `GetCompiledType` 根据指定的 View 文件虚拟路径得到编译后的 `WebPageView` 类型，然后将该类型交给 `ViewPageActivator` 激活一个具体的 `WebPageView` 对象，并调用其 `Render` 方法完成对 View 的最终呈现。`BuildManagerCompiledView` 利用激活的 `WebPageView` 对 View 的呈现实现在抽象方法 `RenderView` 中，而 `Render` 方法仅仅实现了根据 View 文件虚拟路径对 `WebPageView` 的激活，具体的实现可以通过如下的代码片段来体现。

```
public abstract class BuildManagerCompiledView : IView
{
    //其他成员
    public void Render(ViewContext viewContext, TextWriter writer)
    {
        Type viewType = BuildManager.GetCompiledType(ViewPath);
        object instance = null;
        if (null != viewType)
        {
            //controllerContext 字段表示在构造函数中指定的 ControllerContext
            instance = this.ViewPageActivator.Create(
                controllerContext, viewType);
        }
        this.RenderView(viewContext, writer, instance);
    }
    protected abstract void RenderView(ViewContext viewContext,
        TextWriter writer, object instance);
}
```

RazorView

表示 Razor 引擎下的 View 的类型 `RazorView` 直接继承 `BuildManagerCompiledView`。如下面的代码片段所示，它具有额外的三个只读属性属性。`LayoutPath` 表示 View 使用的布局文件的虚拟路径，而 `RunViewStartPages` 和 `ViewStartFileExtensions` 属性与通过“_ViewStart.cshtml”或“_ViewStart.vbhtml”文件定义的开始页面有关，前者表示是否需要执行开始页面，后者表示开始页面文件的扩展名。对于 Razor 引擎默认创建的 `RazorView`，`RunViewStartPages` 属性为 `True`（意味着总是会执行开始页面）。`ViewStartFileExtensions` 属性表示的字符串集合包含两个元素“cshtml”和“vbhtml”。

```
public class RazorView : BuildManagerCompiledView
{
    public RazorView(ControllerContext controllerContext, string viewPath,
        string layoutPath, bool runViewStartPages,
```

```

        IEnumerable<string> viewStartFileExtensions);
public RazorView(ControllerContext controllerContext,
    string viewPath, string layoutPath, bool runViewStartPages,
    IEnumerable<string> viewStartFileExtensions,
    IViewPageActivator viewPageActivator);

protected override void RenderView(ViewContext viewContext,
    TextWriter writer, object instance);

public string          LayoutPath { get; }
public bool            RunViewStartPages { get; }
public IEnumerable<string> ViewStartFileExtensions { get; }
}

```

RazorView 通过实现 RenderView 方法最终完成了对 View 的呈现。方法传入参数 instance 是通过 BuildManagerCompiledView 激活的 View 对象，通过上面的介绍我们知道这是一个空的 WebViewPage<TModel> 对象（默认情况下是通过默认构造函数创建的）。RazorView 在 RenderView 方法中对其进行初始化后调用 ExecutePageHierarchy 方法将整个页面内容呈现出来。RazorView 实现 RenderView 方法的逻辑基本上可以通过如下的代码片段来表示。

```

public class RazorView : BuildManagerCompiledView
{
    //其他成员
    protected override void RenderView(ViewContext viewContext,
        TextWriter writer, object instance)
    {
        WebViewPage page = instance as WebViewPage;
        //初始化 WebViewPage
        Initialize(page);

        //得到表示开启页面的 WebPageRenderingBase 对象
        WebPageRenderingBase startPage;
        if (this.RunViewStartPages)
        {
            startPage = StartPage.GetStartPage(page, "_ViewStart",
                this.ViewStartFileExtensions);
        }
        HttpContextBase httpContext = viewContext.HttpContext;
        page.ExecutePageHierarchy(
            new WebPageContext(viewContext.HttpContext, null, null),
            writer, startPage);
    }
}

```

实例演示：创建一个简单的 RazorView（S807）

为了让读者了解 RazorView 实现 View 呈现的本质，我们按照其实现原理自定义一个简单的 RazorView 类型，在一个 ASP.NET MVC 应用中定义了如下一个表示自定义 RazorView 的 SimpleRazorView 类型，SimpleRazorView 直接实现了 IView 接口，在构造函数中初始化的属性 ViewPath 表示 View 文件的虚拟路径。

```

public class SimpleRazorView: IView
{

```

```

public string ViewPath { get; private set; }

public SimpleRazorView(string viewPath)
{
    this.ViewPath = viewPath;
}

public void Render(ViewContext viewContext, TextWriter writer)
{
    Type viewType = BuildManager.GetCompiledType(this.ViewPath);
    object instance = Activator.CreateInstance(viewType);
    WebViewPage page = (WebViewPage)instance as WebViewPage;

    page.VirtualPath = this.ViewPath;
    page.ViewContext = viewContext;
    page.ViewData = viewContext.ViewData;
    page.InitHelpers();

    WebPageContext pageContext =
        new WebPageContext(viewContext.HttpContext, null, null);
    WebPageRenderingBase startPage = StartPage.GetStartPage(
        page, "_ViewStart", new string[] { "cshtml", "vbhtml" });
    page.ExecutePageHierarchy(pageContext, writer, startPage);
}
}

```

在用于呈现 View 的 Render 方法中，我们利用 BuildManager 根据当前 View 文件的虚拟路径得到动态编译后的类型，然后利用该类型以反射的方式创建一个 WebViewPage 对象。接下来我们初始化该 WebViewPage 对象的 VirtualPath、ViewContext 和 ViewData 属性，并调用 InitHelpers 方法对 HtmlHelper、UrlHelper 和 AjaxHelper 进行初始化。

SimpleRazorView 总是会执行开始页面，所以我们通过调用 System.Web.Mvc.ViewStartPage 的静态方法 GetStartPage 根据指定的开始页面文件名（_ViewStart）和扩展名列表（cshtml 和 vbhtml）得到表示开始页面的 WebPageRenderingBase 对象。最后我们创建 WebPageContext 对象，并将它和表示开始页面的 WebPageRenderingBase 对象作为参数调用 WebViewPage 的 ExecutePageHierarchy 方法实现对整个页面的呈现。

为了验证 SimpleRazorView 能够正常地完成对 View 内容的呈现，我们定义了如下一个 HomeController。在默认的动作方法 Index 中，创建一个 Contact 对象作为当前 ViewData 的 Model，然后通过指定 View 文件的虚拟路径（“~/Views/Home/Index.cshtml”）创建我们自定义的 SimpleRazorView 对象，最后创建 ViewContext 并将其作为参数调用 SimpleRazorView 的 Render 方法将默认的 View 呈现出来。

```

public class HomeController : Controller
{
    public void Index()
    {
        ViewData.Model = new Contact {
            Name = "张三",
            PhoneNo = "123456789",
            EmailAddress = "zhangsan@gmail.com" };
        SimpleRazorView view = new SimpleRazorView("~/Views/Home/Index.cshtml");
    }
}

```

```

        ViewContext viewContext = new ViewContext(ControllorContext, view,
            ViewData, TempData, Response.Output);
        view.Render(viewContext, viewContext.Writer);
    }
}

public class Contact
{
    [DisplayName("姓名")]
    public string Name { get; set; }

    [DisplayName("电话号码")]
    public string PhoneNo { get; set; }

    [DisplayName("电子邮箱地址")]
    public string EmailAddress { get; set; }
}

```

我们的 View 很简单。如下面的代码片段所示，这是一个 Model 类型为 Contact 的强类型 View，在该 View 中直接调用 `HtmlHelper<TModel>` 的扩展方法 `EditorForModel` 将作为 Model 的 Contact 对象以编辑模式呈现在一个表单之中。

```

@model Contact
@{
    ViewBag.Title = Model.Name;
}

@using (Html.BeginForm())
{
    @Html.EditorForModel()
    <input type="submit" value="保存" />
}

```

为了验证我们自定义的 `SimpleRazorView` 对布局文件和 `_ViewStart` 页面的支持，在“~/Views/Shared/”目录下定义了如下一个名为“_Layout.cshtml”的布局文件，布局文件的设置通过定义在“~/Views/”目录下具有如下定义的“_ViewStart.cshtml”文件来指定。

```

_Layout.cshtml:
<html>
  <head>
    <title>@ViewBag.Title </title>
  </head>
  <body>
    <h3>编辑联系人信息</h3>
    @RenderBody()
  </body>
</html>

_ViewStart.cshtml:
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

```

运行程序后会在浏览器中呈现如图 8-14 所示的输出结果，可以看出这和我们直接在 Action 方法 Index 中返回一个 `ViewResult` 对象没有什么不同。

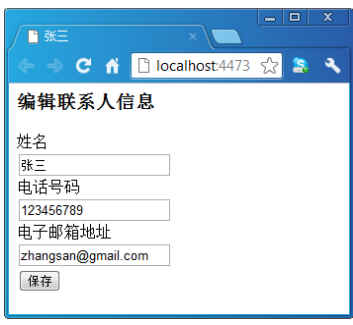


图 8-14 通过自定义 RazorView 呈现出的 View

实例演示：以 IoC 的方式激活 View（S808）

在前面介绍 BuildManagerCompiledView 的时候我们谈到，默认使用的 ViewPageActivator 使用当前注册的 DependencyResolver 来完成对目标 View 的激活，这意味着可以通过注册自定义 DependencyResolver 的方式实现基于 IoC 的 View 激活。在第 3 章“Controller 的激活”中我们创建了一个针对 Ninject 的 DependencyResolver 实现了基于 IoC 的 Controller 激活方式，现在通过这个自定义的 NinjectDependencyResolver 来激活 View。

我们演示的是一个针对多语言支持的场景，为了让 View 上输出的一些内容随着当前线程的 UI Culture 而动态地变化，我们在一个 ASP.NET MVC 应用中定义如下一个读取资源内容抽象类 ResourceReader。这里资源是一个宽泛的概念，并不对存储方式作强制的约束，我们可以使用资源文件也可以使用数据库来存储资源内容，简单起见，ResourceReader 仅仅定义了唯一的 GetString 方法获取指定名称的字符串。

```
public abstract class ResourceReader
{
    public abstract string GetString(string name);
}
```

我们默认采用资源文件来定义数据源，为此在项目中添加了两个资源文件 Resources.resx（语言文化中性）和 Resources.zh.resx（中文），并在资源文件中添加了如图 8-15 所示的资源项（HelloWorld）。

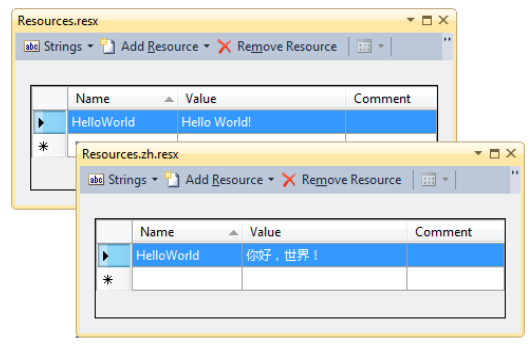


图 8-15 基于不同语言文化的资源文件定义

然后创建如下一个默认的 `DefaultResourceReader`，它默认读取我们添加的资源文件来获取 `GetString` 方法返回的字符串（静态类型 `Resources` 是添加资源文件自动创建的类型）。

```
public class DefaultResourceReader : ResourceReader
{
    public override string GetString(string name)
    {
        return Resources.ResourceManager.GetString(name);
    }
}
```

为了让 `ResourceManager` 能够应用到所有的 `View` 中，我们为整个应用的 `View` 创建了一个基类 `LocalizableViewPage<TModel>`，该类型是 `WebViewPage<TModel>` 的子类，它具有一个类型为 `ResourceManager` 的属性 `ResourceManager`。由于该属性上应用了 `Ninject.InjectAttribute` 特性，意味着该属性会以“属性注入”的方式被自动初始化。

```
public abstract class LocalizableViewPage<TModel>: WebViewPage<TModel>
{
    [Inject]
    public ResourceReader ResourceReader { get; set; }
}
```

接下来我们定义了如下一个简单的 `HomeController`，其默认的动作方法 `Index` 中直接将对应的 `View` 呈现出来。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

如下所示的是动作方法 `Index` 对应 `View` 的定义，使用 `@inherits` 指令让动态编译生成的 `View` 类型继承自我们自定义的基类 `LocalizableViewPage<object>`，直接调用 `ResourceReader` 属性的 `GetString` 方法提取名称为“`HelloWorld`”的字符串资源内容并将其显示出来。

```
@inherits LocalizableViewPage<object>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <h2>@ResourceReader.GetString("HelloWorld")</h2>
    </body>
</html>
```

我们采用基于 URL 的语言文化决定机制，即将语言文化的代码置于请求 URL 中来决定希望采用的语言。为此我们在自动生成的 `RouteConfig` 类型中注册了如下一个 URL 模板为“`{culture}/{controller}/{action}`”的路由对象。

```
public class RouteConfig
{
```

```

public static void RegisterRoutes(RouteCollection routes)
{
    //其他操作
    routes.MapRoute(
        name       : "Default",
        url        : "{culture}/{controller}/{action}",
        defaults   : new{
            culture    = "zh-CN",
            controller = "Home",
            action     = "Index"
        }
    );
}
}

```

我们自定义的 `DefaultResourceReader` 能够根据当前线程的 `UICulture` 选择对应的资源文件,那么我们需要根据请求地址指示的语言文件对当前线程的语言文件进行相应的设置即可。于是在 `Global.asax` 中定义了如下一个 `Application_BeginRequest` 方法,它会在 `HttpApplication` 的 `BeginRequest` 事件触发的时候被执行并从请求地址中提取语言文化代码,然后对当前线程的语言文化进行相应的设置。除此之外,针对 `NinjectDependencyResolver` 的注册和 `ResourceReader` 与 `Default ResourceReader` 之间的映射关系定义在 `Application_Start` 方法中。

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        NinjectDependencyResolver dependencyResovler =
            new NinjectDependencyResolver();
        dependencyResovler.Register<ResourceReader, DefaultResourceReader>();
        DependencyResolver.SetResolver(dependencyResovler);
    }

    protected void Application_BeginRequest()
    {
        HttpContextBase contextWrapper =
            new HttpContextWrapper(HttpContext.Current);
        string culture = RouteTable.Routes.GetRouteData(contextWrapper)
            .Values["culture"] as string;
        if (!string.IsNullOrEmpty(culture))
        {
            try
            {
                CultureInfo cultureInfo = new CultureInfo(culture);
                Thread.CurrentThread.CurrentCulture = cultureInfo;
                Thread.CurrentThread.CurrentUICulture = cultureInfo;
            }
            catch {}
        }
    }
}

```

现在运行程序,并通过地址指定采用的语言文化,可以发现呈选出来的内容与我们指定的语言文化是一致的,具体的输出效果如图 8-16 所示。

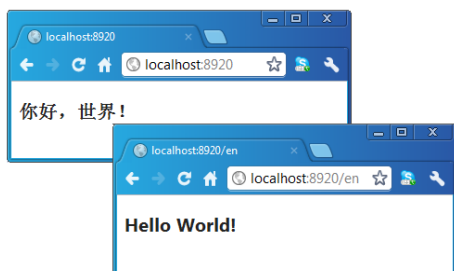


图 8-16 通过请求 URL 决定语言的选择

8.3.4 RazorViewEngine

基于 Web Form 引擎的 `System.Web.Mvc.WebFormViewEngine` 和针对 Razor 引擎的 `System.Web.Mvc.RazorViewEngine` 都是抽象类型 `System.Web.Mvc.BuildManagerViewEngine` 的子类, 而后者又继承自 `System.Web.Mvc.VirtualPathProviderViewEngine`, 在这里我们仅仅对实现在 `RazorViewEngine` 中 View 获取的逻辑进行简单介绍。

由于 Razor 引擎下的 View 通过 `RazorView` 对象来表示, 而 `RazorView` 通过 View 文件的虚拟路径来构建, 所以 `RazorViewEngine` 的 View 获取机制在于根据当前上下文找到与指定 View 名称相匹配的 View 文件 (.cshtml 或者 .vbhtml 文件), 然后根据该 View 文件的虚拟路径创建一个 `RazorView` 对象并最终封装成 `ViewEngineResult` 对象返回。

实现在 `RazorViewEngine` 中的目标 View 文件的搜索是根据一个预定义顺序进行的, 如果当前请求不是针对某个 Area 的, 下面的列表代表了 View 的搜索顺序。

- `~/Views/{ControllerName}/{ViewName}.cshtml`
- `~/Views/{ControllerName}/{ViewName}.vbhtml`
- `~/Views/Shared/{ViewName}.cshtml`
- `~/Views/Shared/{ViewName}.vbhtml`

对于针对某个 Area 的请求, `RazorViewEngine` 会先按照如下的顺序对目标 View 进行搜索。如果在这个列表中没有成功找到目标 View 文件, 会继续按照上面的属性进行搜索。

- `~/Areas/{AreaName}/Views/{ControllerName}/{ViewName}.cshtml`
- `~/Areas/{AreaName}/Views/{ControllerName}/{ViewName}.vbhtml`
- `~/Areas/{AreaName}/Views/Shared/{ViewName}.cshtml`
- `~/Areas/{AreaName}/Views/Shared/{ViewName}.vbhtml`

如果按照上面的搜索顺序依然找不到目标 View 文件, `RazorViewEngine` 会根据这个列表创建并返回一个 `ViewEngineResult` 对象。这里介绍的 View 搜索机制不仅仅应用于普通的 View 文件, 还应用于 Partial View 和布局文件的搜索。

ViewEngine 不仅仅通过 FindView/FindPartialView 根据当前上下文获取指定的 View，还通过 ReleaseView 对指定的 View 进行释放回收操作。ReleaseView 方法在 RazorViewEngine 的实现很简单，如果指定的 View 对象的类型实现 IDispose 接口，它会直接调用其 Dispose 方法。图 8-17 所示的 UML 体现了 Razor 引擎涉及的相关类型/接口以及它们之间的相互关系。

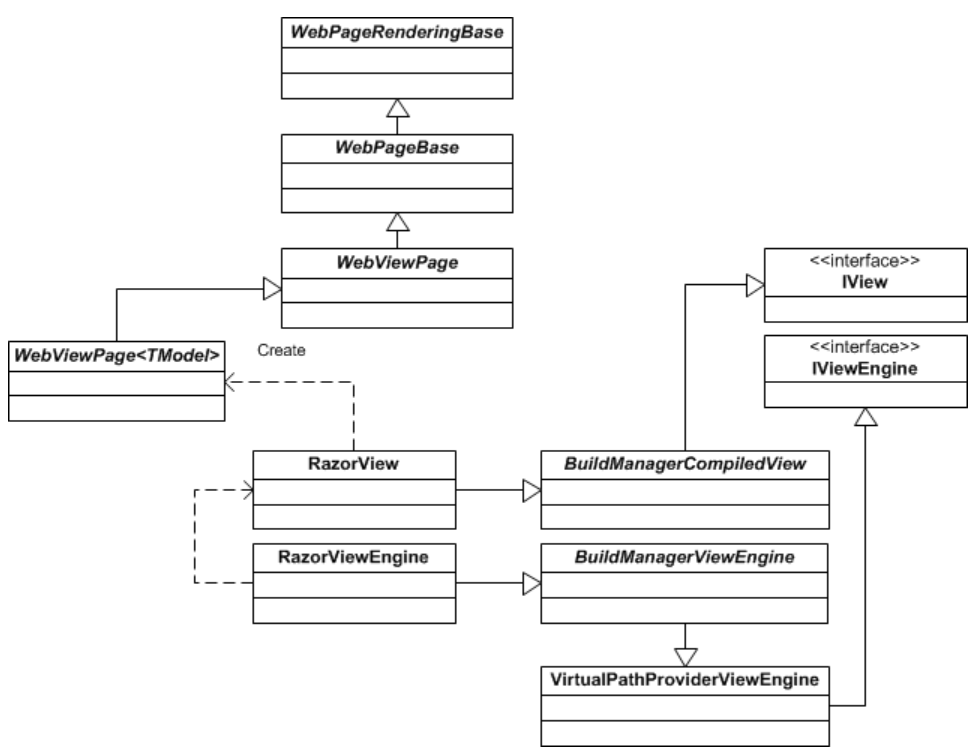


图 8-17 Razor 引擎涉及的相关类型/接口

实例演示：创建一个简单的 RazorViewEngine (S809)

在前面的实例 (S807) 演示中我们创建了一个用于模拟 RazorView 的 SimpleRazorView，现在为它创建一个对应的 RazorViewEngine，直接在该实例项目中添加如下一个 SimpleRazorViewEngine。

```
public class SimpleRazorViewEngine: IViewEngine
{
    private string[] viewLocationFormats = new string[] {
        "~/Views/{1}/{0}.cshtml",
        "~/Views/{1}/{0}.vbhtml",
        "~/Views/Shared/{0}.cshtml",
        "~/Views/Shared/{0}.vbhtml" };
    private string[] areaViewLocationFormats = new string[] {
        "~/Areas/{2}/Views/{1}/{0}.cshtml",
        "~/Areas/{2}/Views/{1}/{0}.vbhtml",
        "~/Areas/{2}/Views/Shared/{0}.cshtml",
        "~/Areas/{2}/Views/Shared/{0}.vbhtml" };
}
```

```

public ViewEngineResult FindPartialView(ControllerContext controllerContext,
    string partialViewName, bool useCache)
{
    return FindView(controllerContext, partialViewName, null, useCache);
}

public ViewEngineResult FindView(ControllerContext controllerContext,
    string viewName, string masterName, bool useCache)
{
    string controllerName = controllerContext.RouteData
        .GetRequiredString("controller");
    object areaName;
    List<string> viewLocations = new List<string>();
    Array.ForEach(viewLocationFormats, format =>
        viewLocations.Add(string.Format(format, viewName, controllerName)));
    if (controllerContext.RouteData.Values.TryGetValue("area", out areaName))
    {
        Array.ForEach(areaViewLocationFormats,
            format => viewLocations.Add(string.Format(format, viewName,
                controllerName, areaName)));
    }
    foreach (string viewLocation in viewLocations)
    {
        string filePath = controllerContext.HttpContext.Request
            .MapPath(viewLocation);
        if (File.Exists(filePath))
        {
            return new ViewEngineResult(new SimpleRazorView(viewLocation),
                this);
        }
    }
    return new ViewEngineResult(viewLocations);
}

public void ReleaseView(ControllerContext controllerContext, IView view)
{
    IDisposable disposable = view as IDisposable;
    if (null != disposable)
    {
        disposable.Dispose();
    }
}
}

```

我们完全按照上面介绍的路径顺序搜索指定的目标 View。简单起见，在对目标 View 进行搜索时忽略了指定的布局文件名和对 ViewEngineResult 的缓存。这个自定义的 SimpleRazorViewEngine 在 Global.asax 中通过如下的代码对进行注册。

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        ViewEngines.Engines.Clear();
        ViewEngines.Engines.Add(new SimpleRazorViewEngine());
    }
}

```

然后我们按照正常的方式来定义 HomeController 的 Action 方法 Index，运行程序之后依然会得到如图 8-14 所示的输出结果。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        Contact contact = new Contact
        {
            Name          = "张三",
            PhoneNo       = "123456789",
            EmailAddress  = "zhangsan@gmail.com"
        };
        return View(contact);
    }
}
```

本章小结

ASP.NET MVC 针对请求的处理最终体现在对目标 Action 的执行，并就处理的结果对请求予以响应，而 ActionResult 为我们提供了一种响应请求的快捷方式。ASP.NET MVC 定义了一系列的原生的 ActionResult 类型，其中包括 EmptyResult、ContentResult、FileResult、JavaScriptResult、JsonResult、HttpStatusCodeResult、RedirectResult 和 RedirectToRouteResult 等，它们或者帮助我们将指定的内容按照相应的媒体类型响应给客户端，或者回复一个指定状态码的响应，又或者实现客户端的重定向。

ViewResult 是重要也是最为常用的 ActionResult，我们可以利用它将指定的 View 呈现在客户端的浏览器上。针对 ViewResult 的 View 呈现最终是利用 View 引擎实现的。View 引擎中的 View 实现了 IView 接口，对应着某个 View 文件，而核心组件 ViewEngine 实现针对 View 的获取、激活、呈现以及最终的释放。

ASP.NET MVC 提供了 Web Form 和 Razor 两种 View 引擎。Razor 引擎中的 View 和 ViewEngine 对应的类型为 RazorView 和 RazorViewEngine。RazorView 对应一个以 .cshtml/.vbhtml 文件定义的 View 文件，这样的文件通过 ASP.NET 的动态编译生成一个 WebViewPage<TModel> 类型。RazorView 通过激活的 WebViewPage<TModel> 对象实现了对 View 的最终呈现。

第9章 ASP.NET Web API

不知道大家是否注意到一个现象，传统 Web 服务其实并没有直接建立在 Web 上而是建立在 SOAP 上，以至于我们提到 Web 服务就会想到 SOAP。随着在 Web 服务越来越多地采用 REST 架构，SOAP 在整个 Web 服务体系中的垄断地位正在发生改变，或者已经发生了改变。REST 提倡一种面向资源的架构，直接在 Web 上建立一种轻量级的 Web 服务。虽然 WCF 自 3.5 之后提供了针对 REST 的支持，但是这种在“重量级”通信平台上实现的“轻量级”消息通信给我们一种“牛刀杀鸡”之感，所以直接建立在 ASP.NET 平台上的 ASP.NET Web API 是更好的选择。

9.1 Web、REST 与 Web API

如果说在过去半个世纪中哪种信息技术对人类影响最为深远，我想很多人的答案是 WWW（World Wide Web、W3 或者万维网），它完全改变了我们的生活和思维方式。WWW 重要性甚至可以从 W3C 对它的定义中看出来：“The World Wide Web is the universe of network-accessible information, an embodiment of human knowledge.”（万维网是信息的来源、知识的化身）。

9.1.1 Web 如此简单

Web 的简单性使之能够得到广泛的普及，并且成为互联网的标准。它由 URI、HTTP 和 HTML 三个基本的标准构成。HTTP 是 Web 的核心，就是一个简单的请求—回复的传输协议，客户端请求什么，服务端就给什么，并且每次消息交换均是独立的。HTTP 是一种文档化的协议（Documented Protocol），客户端将请求文档置于 HTTP 请求封套（Envelope）中发送给服务端，而服务端将响应文档置于 HTTP 响应封套中返回给客户端。

```
GET http://www.microsoft.com/en-us/default.aspx HTTP/1.1
Host: www.microsoft.com
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/535.7 (KHTML, like Gecko)
Chrome/16.0.912.75 Safari/535.7
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: ...
```

上面这个文本片段反映的是我们通过 Chrome 浏览器访问微软的官网（www.microsoft.com）对应的 HTTP 请求，我们借此来简单谈谈作为 HTTP 请求封套的基本结构。第一行体现了 HTTP 请求的三个基本属性，即 HTTP 方法（GET）、目标资源（http://www.microsoft.com/en-us/default.aspx）和协议版本（HTTP/1.1）。

HTTP 方法（HTTP Method 或者 HTTP Verb）对于 REST 来说是一个非常重要的概念。如果将请求的目标内容视为一种网络资源的话，HTTP 方法反映了针对该资源的操作类型。我们常用的几种 HTTP 方法（比如 GET/HEAD、PUT、POST 和 DELETE）分别体现了针对目标资源的获取、添加、修改和删除操作。

除首行文字之外的所有内容被称为 HTTP 请求报头（Header），而每个报头实际上是一个键—值对。HTTP 协议本身定义一系列原生的报头，我们也可以根据需要在报头集合中添加任意的请求报头。上面给出的 HTTP 请求中包含了一些常用的请求报头，我们可以根据它们获得主机名称、采用的缓存策略、浏览器相关信息，以及客户端支持的媒体类型（Media

Type)、编码方式、语言和字符集等。

除了请求报头集合之外，一个 HTTP 请求可以具有相应的主体内容。一个 HTTP 请求的主体可以具有任意的格式，比如以 XML 或者 JSON 表示的请求数据，或者是一个结构化的 HTML 文档（比如一个包含提交表单的 HTML 文档来向服务端提供用户在浏览器上输入的内容）。

```

HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/7.5
X-AspNet-Version: 2.0.50727
VTag: 791897542300000000
P3P: CP="ALL IND DSP COR ADM CONo CUR CUSo IVAo IVDo PSA PSD TAI TELo OUR SAMo
CNT COM INT NAV ONL PHY PRE PUR UNI"
X-Powered-By: ASP.NET
Date: Wed, 18 Jan 2012 07:06:25 GMT
Content-Length: 34237

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>...</html>

```

前面的 HTTP 请求通过浏览器发送给服务端之后会接收到具有如上内容的 HTTP 响应，该消息的第 1 行表示采用的协议版本和响应 HTTP 状态码/文字描述。“200 OK”表示请求被正常接收处理，常见的响应状态码包括“401，Not Authorized”（授权失败）和“404，Not Found”（请求资源不存在）等。

与 HTTP 请求消息一样，HTTP 响应消息依然具有一组报头，而响应的内容被封装到 HTTP 响应消息的主体部分。响应内容的格式（比如 XML、JSON、HTML 等）一般与表示媒体类型的响应报头“Content-Type”（比如“application/xml”、“application/json”和“text/html”等）一致。上面这个 HTTP 响应的主体内容为 HTML 文档，所以“Content-Type”报头表示的媒体类型为“text/html”。

9.1.2 REST 是什么

REST 与技术无关，代表的是一种软件架构风格。REST 是 Representational State Transfer 的简称，中文翻译为“表征状态转移”，Roy Fielding 博士（他同时也是 HTTP 的制定者之一）于 2000 年在其博士论文 *Architectural Styles and the Design of Network-based Software Architectures*（<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>）中最早提出这种 Web 服务的架构风格，在这之前 Web 服务具有两种主流的架构风格，即 SOAP 和 XML-RPC。

REST 从资源的角度来审视整个网络，它将分布在网络中某个节点中的资源通过 URI

进行标识, 客户端应用通过 URI 来获取资源的表征, 获得这些表征致使这些应用程序转变了状态。随着不断获取资源的表征, 客户端应用不断地在转变着状态。

为什么会起这么一个名字呢? Roy Fielding 是这样解释的: “设计良好的网络应用表现为一系列的网页, 这些网页可以看作虚拟的状态机, 用户选择这些链接导致下一网页传输到用户端展现给使用的人, 而这正代表了状态的转变。”这貌似在说 Web 站点而不是 Web 服务, 两者之间的不同之处在于消费者的不同, Web 站点的消费者是人, 而 Web 服务的客户端是机器(应用)。不过我们完全可以将两者统一起来, 采用相同的架构风格来构架这两种不同的 Web 应用。

虽然说作为架构风格的 REST 与具体网络协议无关, 但是我们所说的绝大部分基于 REST 的 Web 服务都是建立在 HTTP 之上的。REST 通常使用 HTTP、URI、XML 以及 HTML 这些广泛采用的协议和标准。对于 REST 这种面向资源的架构风格, 有人甚至提出了一种全新的架构理念, 即面向资源架构 (ROA: Resource Oriented Architecture), 现在我们简单地讨论一下关于 REST/ROA 的一些基本的设计原则。

一切数据都是资源

所有的数据, 不论是通过网络请求获取的还是操作(创建、修改和删除)的数据, 都是资源。这不仅仅包括图片、MP3 和视频这些通过具体文件承载的物理资源, 还包括通过关系型数据库保存的数据, 甚至包括一些经过实时计算得到的数据。将一切数据视为资源是 REST 区别于其他架构风格的最为本质的属性。

所有的资源均可被唯一标识

资源能够通过网络访问的形式被获取和操作, 不仅仅要求它们是可被寻址 (Addressable) 的, 还要求它们能够被唯一标识。我们通过 URI 来唯一标识某个资源, 如下的列表表示某个 CRM 服务标识某个或者某类客户的 URI。

- 所有的客户: <http://www.artech.com/CRMService/Customers>
- 所有金牌客户: <http://www.artech.com/CRMService/Customers/Gold>
- 一个编码 C001 的金牌客户: <http://www.artech.com/CRMService/Customers/Gold/C001>

采用统一而简单的接口

一个基于 REST/ROA 的 Web 服务操作请求只需要体现两点, 即资源的唯一标识和操作类型。资源通过 URI 来标识, 而操作类型则可以通过 HTTP 方法 (GET/HEAD、POST、PUT 和 DELETE 等) 来体现。这和基于 SOAP 的架构风格完全不同, 后者实际上是通过 SOAP 消息的 <Action> 报头来进行操作识别的。同样以上面提到的 CRM 服务为例, 如果我们需要分别针对某个单一客户进行获取、添加、修改和删除操作, 其 <Action> 报头值可能被定义成如下的形式。

- `http://CRMService/GetCustomer`
- `http://CRMService/AddCustomer`
- `http://CRMService/UpdateCustomer`
- `http://CRMService/DeleteCustomer`

如果采用 REST 风格来构建 CRMService 这个 Web 服务，请求 URI 就表示被操作的某个客户端，而四种基本的操作类型则通过相应的 HTTP 方法来体现。比如我们要操作一个编号为 C001 的客户，可以采用如下具有相同 URL 但是具有不同 HTTP 方法的请求实现对该客户的获取、添加、修改和删除等操作。

- `http://CRMService/Customers/C001 + GET`
- `http://CRMService/Customers/C001 + PUT1`
- `http://CRMService/Customers/C001 + POST`
- `http://CRMService/Customers/C001 + DELETE`

基于表征的通信

REST 的首字母 R (Representational) 可以理解为对资源一种“表征”，无论是作为请求还是响应的数据都是对相应资源的表征。由于 REST 主要面向 Web，我们倾向于直接采用 XML 或者 JSON 来表示被操作的资源。

```
public class Customer
{
    public Id {get;set;}
    public Name {get;set;}
    public string Province {get;set;}
    public string City {get;set;}
}
```

比如对于通过 C# 定义的数据类型 Customer，我们可以直接通过如下所示的 XML 或者 JSON 片段来表示某个 Customer 对象。

```
XML:
<Customer>
  <Id>C001</Id>
  <Name>张三</Name>
  <Province>江苏</Province>
  <City>苏州</City>
</Customer>

JSON:
{
    Id      : 001,
```

¹ 采用 HTTP-PUT 方法的 HTTP 请求执行客户添加操作时，请求地址可以不需要指定添加客户的 ID，因为包含添加客户端数据的主体部分往往包含对应的 ID。对于执行客户修改操作的 HTTP-POST 请求亦是如此。

```
Name      : 张三,  
Province  : 江苏,  
City      : 苏州  
}
```

无状态服务调用

服务调用采用完全独立的消息交换方式，不需要为客户端保持会话状态，这不仅仅是为了“迎合”HTTP 无状态的特性，同时赋予了服务较强的可伸缩性。对于无状态的 Web 服务，负载均衡（Load Balance）变得很容易。

9.1.3 ASP.NET Web API

ASP.NET Web API 直接借鉴了 ASP.NET MVC 的设计，所以两者具有非常类似的编程模式。我们以 Controller 的形式来定义服务，而 Controller 中的 Action 方法则代表具体的操作。接下来通过实例演示的方式来介绍如何定义和调用 Web API。这个实例是一个单页的 Web 应用，模拟对联系人数据的 CRUD 操作。从如图 9-1 所示的应用截图可以看到，Web 页面中会显示所有联系人列表，针对联系人的添加、修改和删除都在同一个页面中完成。（S901）

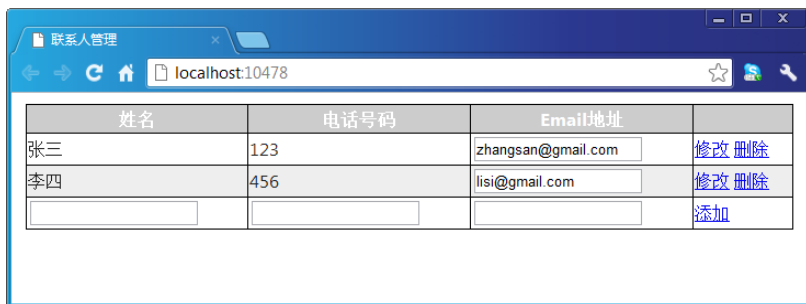


图 9-1 “联系人管理器”应用截图

针对联系人信息的 CRUD 操作都是通过调用 Web API 的形式来完成的。具体来说，我们利用了 jQuery 以 Ajax 的方式调用 Web API。至于界面的设计，我们充分利用了 ASP.NET MVC 提供的另一个名为 KnockOut 的 JavaScript 框架实现了对数据的绑定，也就是说，整个应用基本上只涉及前端的 HTML/JavaScript 和后端的 Web API 两个部分。

HttpController

我们现在就来简单地介绍这个作为“联系人管理器”的 Web 应用是如何创建的，先在一个 ASP.NET MVC 应用中定义了如下一个表示联系人的 Contact 类型。

```
public class Contact
{
    public string Id { get; set; }
    public string Name { get; set; }
    public string PhoneNo { get; set; }
    public string EmailAddress { get; set; }
}
```

然后在 **Controllers** 目录下定义如下一个表示联系人管理服务的 **ContactsController**，它的基类不再是我们熟悉的抽象类 **Controller**，而是 **System.Web.Http.ApiController**。**ApiController** 实现了接口 **System.Web.Http.Controllers.IHttpController**，所以在本章后续部分将 **ASP.NET Web API** 中的 **Controller** 称为 **HttpController**。

```
public class ContactsController : ApiController
{
    private static List<Contact> contacts = new List<Contact>
    {
        new Contact
        {
            Id          = "001",
            Name        = "张三",
            PhoneNo     = "123",
            EmailAddress = "zhangsan@gmail.com"
        },
        new Contact
        {
            Id          = "002",
            Name        = "李四",
            PhoneNo     = "456",
            EmailAddress = "lisi@gmail.com"
        }
    };

    public IEnumerable<Contact> Get()
    {
        return contacts;
    }

    public Contact Get(string id)
    {
        return contacts.FirstOrDefault(c => c.Id == id);
    }

    public void Put(Contact contact)
    {
        contact.Id = Guid.NewGuid().ToString();
        contacts.Add(contact);
    }

    public void Post(Contact contact)
    {
        Delete(contact.Id);
        contacts.Add(contact);
    }
}
```

```

public void Delete(string id)
{
    Contact contact = contacts.FirstOrDefault(c => c.Id == id);
    contacts.Remove(contact);
}
}

```

我们直接利用一个静态 `Contact` 列表字段来表示数据存储，针对联系人的所有操作都是针对这个列表进行的。对于代表 CRUD 操作的五个 Action 方法，我们直接采用相应的 HTTP 方法（Put、Get、Post 和 Delete）来对其进行命名。如果 Web API 根据 URL 路由不能得到目标 Action 的名称，它会将当前请求的 HTTP 方法作为前缀并借助于得到的路由数据找到匹配的 Action 方法。

路由注册

与 ASP.NET MVC 一样，Web API 同样通过 URL 路由机制根据请求的地址得到需要激活的 ApiController 和对应的 Action 名称（只有在 URL 路由不能解析出 Action 名称的情况下才会根据 HTTP 方法对目标 Action 进行选择）。当我们利用 Visual Studio 提供的向导创建一个创建 ASP.NET Web API 或者 ASP.NET MVC 应用的时候，在自动生成的静态类型 WebApiConfig 中会默认为我们注册如下一个路由。

```

public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}

```

如上面的代码片段所示，针对 Web API 路由的注册通过调用 RouteCollection 的扩展方法 MapHttpRoute 实现的，它注册的是一个 System.Web.Http.WebHost.Routing.HttpWebRoute 对象，该类型定义在程序集 System.Web.Http.WebHost 中。默认的 URL 模板为“api/{controller}/{id}”，针对 Action 名称的路由变量并没有包含在模板中，所以最终针对目标 Action 的选择是根据 HTTP 方法完成的。

根据 HTTP 方法与 Action 名称的匹配机制，如果我们通过浏览器访问地址“/api/contacts”或者“api/contacts/001”，用于返回所有联系人列表的 Get 方法和返回指定联系人信息的 Get 方法会被视为匹配的 Action 方法被执行。如图 9-2 所示，当我们通过 Chrome 来访问这两个地址时，返回的联系人会以 XML 的形式直接显示在浏览器上。

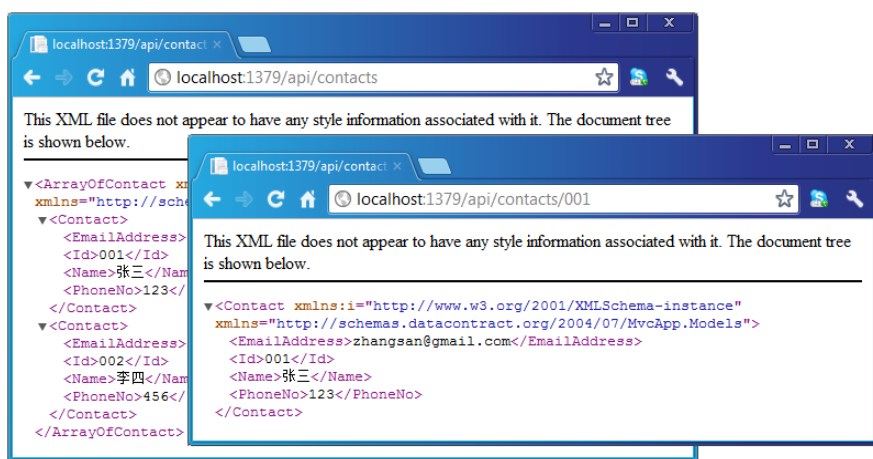


图 9-2 通过浏览器调用 Web API (Chrome)

有人可能会问这样一个问题,为什么通过浏览器发起对 Web API 的调用得到的数据是以 XML 格式而不是以 JSON 的方式被序列化的呢?实际上 ASP.NET Web API 对数据对象的序列化是非常智能的,它能够根据请求希望的媒体类型选择适合的序列化方式。在 ASP.NET Web API 中对请求/响应进行序列化/反序列化的序列化器是与某种媒体类型关联的,它能够根据请求消息携带的媒体类型“智能地”选择匹配的序列化器。

具体来说,ASP.NET Web API 会提取 HTTP 请求的“Accept”报头得到客户端可以接受的媒体类型列表,按照先后顺序(从左到右)去获取匹配的序列化器,优先匹配的序列化器会被使用。如果没有匹配的序列化器,默认使用的是针对 JSON 的序列化器。

如下所示的是通过 Chrome 访问地址“/api/contacts/001”的请求消息,其“Accept”报头携带的媒体类型列表为“text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8”,最终导致采用 XML 序列化方式的是源于媒体类型“application/xml”被优先匹配。

```
GET http://localhost:1379/api/contacts/001 HTTP/1.1
Host: localhost:1379
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/536.11 (KHTML, like Gecko)
Chrome/20.0.1132.47 Safari/536.11
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: AJSTAT_ok_times=3; _ourplusFirstTime=112-5-15-14-28-38;
_ourplusReturnCount=13; _ourplusReturnTime=112-5-15-14-41-25; theme=Theme2
```

如果我们通过 IE 来访问相同的地址(“/api/contacts/001”),则会生成具有如下内容的请求消息。可以清楚地看到 Accept 报头的媒体列表中不再包含“application/xml”,这会导致 Web API 找不到与媒体类型完全匹配的序列化器,所以最终会选择基于 JSON 的默认序列化器。

```
GET http://localhost:1379/api/contacts/001 HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
Host: localhost:1379
```

在默认情况下, IE 并不会将格式化为 JSON 的联系人信息显示在浏览器中, 而是会弹出一个对话框提示我们将内容保存为一个文件, 或者选择相应的应用程序查看响应的内容。如下所示的是整个 HTTP 响应的内容, 可以看到响应报头“Content-Type”的值为“application/json; charset=utf-8”, 而主体内容正是 Contact 对象的 JSON 表示。

```
HTTP/1.1 200 OK
Server: ASP.NET Development Server/10.0.0.0
Date: Sat, 07 Jul 2012 08:05:07 GMT
X-AspNet-Version: 4.0.30319
Cache-Control: no-cache
Pragma: no-cache
Expires: -1
Content-Type: application/json; charset=utf-8
Content-Length: 80
Connection: Close

{"Id": "001", "Name": "张三", "PhoneNo": "123", "EmailAddress": "zhangsan@gmail.com"}
```

Web API 调用

因为 Web API 完全是基于 Web 的, 所以针对它的调用本质上就是一个简单的 HTTP 请求和响应过程, 可以通过手工地发送请求接收响应的方式来进行 Web API 的调用, 也可以通过 Ajax 的方式调用 Web API。我们的演示实例采用的是基于 jQuery 的 Ajax 调用, 由于客户端直接通过 Ajax 与 Web API 进行交互, 所以本实例根本不涉及任何的后台代码, 如下面的代码片段所示, 定义在 HomeController 的 Action 方法 Index 仅仅是将对应的 View 呈现出来而已。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

如下所示的是 Action 方法 Index 对应 View 的定义, ASP.NET MVC 框架本身提供了两个 JavaScript 框架, 其中一个是 jQuery, 另一个是 Knockout (以下简称 KO)。单表实例应用唯一页面的这个 View 利用 KO 实现了对数据的绑定和与用户交互处理。

```
<html>
  <head>
    <title>联系人管理</title>
    <script type="text/javascript"
      src="../../Scripts/jquery-1.7.1.js"></script>
```

```

    <script type="text/javascript"
        src="../../Scripts/knockout-2.1.0.js"></script>
</head>
<body>
<div id="contacts">
    <table>
        <tr>
            <th>姓名</th>
            <th>电话号码</th>
            <th>Email 地址</th>
            <th></th>
        </tr>
        <tbody>
            <!-- ko foreach: allContacts-->
            <tr>
                <td data-bind="text: Name" />
                <td data-bind="text: PhoneNo" />
                <td>
                    <input type="text" class="textbox long"
                        data-bind="value: EmailAddress"/>
                </td>
                <td>
                    <a href="#" data-bind="click: $root.updateContact">修改
                    </a>
                    <a href="#" data-bind="click: $root.deleteContact">删除
                    </a>
                </td>
            </tr>
            <!-- /ko -->
            <tr data-bind="with: addedContact">
                <td>
                    <input type="text" class="textbox"
                        data-bind="value: Name"/>
                </td>
                <td>
                    <input type="text" class="textbox"
                        data-bind="value: PhoneNo"/>
                </td>
                <td>
                    <input type="text" class="textbox long"
                        data-bind="value: EmailAddress"/>
                </td>
                <td>
                    <a href="#" data-bind="click: $root.addContact">添加
                    </a>
                </td>
            </tr>
        </tbody>
    </table>
</div>
<script type="text/javascript" >
    function ContactViewModel() {
        self = this;
        self.allContacts = ko.observableArray();
        self.addedContact = ko.observable();

        //加载联系人列表
        self.loadContacts = function () {

```

```

        $.get("/api/contacts", null, function (data) {
            self.allContacts(data);
            var emptyContact = { Id: "", Name: "", PhoneNo: "",
                                EmailAddress: "" };
            self.addedContact(emptyContact);
        });
    };

    //添加联系人
    self.addContact = function (data) {
        if (!self.validate(data)) {
            return;
        }
        $.ajax({
            url: "/api/contacts/",
            data: self.addedContact(),
            type: "PUT",
            success: self.loadContacts
        });
    };

    //修改联系人信息
    self.updateContact = function (data) {
        $.ajax({
            url: "/api/contacts/",
            data: data,
            type: "POST",
            success: self.loadContacts
        });
    };

    //删除联系人
    self.deleteContact = function (data) {
        $.ajax({
            url: "/api/contacts/" + data.Id,
            type: "DELETE",
            success: self.loadContacts
        });
    };

    self.validate = function (data) {
        if (data.Name && data.PhoneNo && data.EmailAddress) {
            return true;
        }
        alert("请输入完整联系人信息!");
        return false;
    }
    self.loadContacts();
}
ko.applyBindings(new ContactViewModel());
</script>
</body>
</html>

```

考虑到可能有人对 KO 这个 JavaScript 框架不太熟悉,在这里我们作一下概括性的介绍。KO 是微软将应用于 WPF/Silverlight 的 MVVM 模式在 Web 上的尝试,这是一个非常有用的 JavaScript 框架。KO 的核心就是绑定,包括数据绑定和行为绑定。对于数据绑定, KO 既支

持单向绑定也支持双向绑定。如果我们将数据绑定到某个表单元素上,对于单向绑定来说,用户在表单元素上输入的值不能引起数据源的改变,而双向绑定则会。所谓的行为绑定,说白了就是将某个函数注册到某个元素的某个事件上。

我们通过一个简单的例子来说明两种绑定在 KO 中的实现。假设需要设计如图 9-3 所示的“地址编辑器页面”,在页面加载的时候它会将默认的地址信息绑定到表示省、市、区和街道的文本框和显示完整地址信息的元素上,当用户在文本框中输入新的值并点击“确认”按钮后,显示的完整地址会相应的变化。



图 9-3 地址编辑器页面

可以利用 KO 按照如下的方式来实现地址信息的绑定和处理用户提交的编辑确认请求,我们需要通过一个函数来创建表示 View Model 的“类”,需要绑定的数据和函数定义在该类的字段中,而组成 View 的 HTML 元素通过内联的 data-bind 属性实现数据绑定和事件注册。我们最终需要创建 View Model 对象,并将其作为参数调用 ko.applyBindings 方法将绑定应用到当前页面。

```
<div>
  <div><label>省:</label><input data-bind="value: province" /></div>
  <div><label>市:</label><input data-bind="value: city" /></div>
  <div><label>区:</label><input data-bind="value: district" /></div>
  <div><label>街道:</label><input data-bind="value: street"/>
  <div><label>地址:</label><span data-bind="text: address"></span></div>
  <div><input type="button" data-bind="click: format" value="确定"/></div>
</div>

<script type="text/javascript" >
  function AddressModel() {
    var self = this;
    self.province = ko.observable("江苏省");
    self.city = ko.observable("苏州市");
    self.district = ko.observable("工业园区");
    self.street = ko.observable("星湖街 328 号");
    self.address = ko.observable();

    self.format = function () {
      if (self.province() && self.city() && self.district() && self.street()) {
        var address = self.province() + " " +
          self.city() + " " + self.district() + " " + self.street();
      }
    }
  }
</script>
```

```

        self.address(address);
    }
    else {
        alert("请提供完整的地址信息");
    }
};
self.format();
}
ko.applyBindings(new AddressModel());
</script>

```

如上面的代码片段所示，我们定义了一个名为 `AddressModel` 的函数作为整个“地址编辑”页面的 View Model，`AddressModel` 的五个数据成员（`province`、`city`、`district`、`street` 和 `address`）表示地址的四个组成部分和格式化的地址，它们都是基于双向绑定的 `Observable` 类型成员，意味着用户的输入能够即时改变绑定的数据源，而数据源的改变也能即时地反映在绑定的 HTML 元素上。`Observable` 数据成员是一个通过调用 `ko.observable` 方法创建的函数，方法调用指定的参数表示更新的数据。

`AddressModel` 的另一个成员 `format` 是一个自定义的函数，该函数进行地址格式化并用格式化的地址更新 `address` 字段。由于 `address` 字段是一个 `Observable` 成员，一旦它的值发生改变，被绑定的 HTML 元素的值将会自动更新。

`AddressModel` 的六个字段分别绑定在六个 HTML 元素上，其中 `province`、`city`、`district` 和 `street` 字段绑定到代表对应文本框的 `Value` 属性上（`data-bind="value: {成员名称}"`），而 `address` 字段则绑定到用于显示格式化地址的 `` 元素的 `Text` 属性上（`data-bind="text: {成员名称}"`），用于格式化地址的 `format` 字段则与“确定”按钮的 `click` 事件进行绑定（`data-bind="click: {成员名称}"`）。真正的绑定工作发生在 `ko.applyBindings` 方法被调用的时候。

在对 KO 有了基本了解的基础上再来看“管理联系人”页面的 HTML 和 JavaScript，发现它们就很好理解了。在该页面中作为 View Model 是我们定义的 `ContactViewModel` 函数，它具有两个数据成员 `allContacts` 和 `addedContact`，分别表示整个联系人列表和添加的联系人，另外四个字段 `loadContacts`、`addContact`、`updateContact` 和 `deleteContact` 分别以 Ajax 的形式调用我们之前定义的 Web API 实现针对联系人的获取、添加、修改和删除操作。

通过调用 `loadContacts` 获取的联系人列表 `allContacts` 以 `foreach` 绑定（`<!-- ko foreach: contacts --><!-- /ko -->`）的形式绑定到 `<table>` 元素相应的行（`<tr>`）中，而表示添加联系人的 `addedContacts` 绑定到 `<table>` 的最后一行上。用于进行添加、修改和删除的 `addContact`、`updateContact` 和 `deleteContact` 则注册到相应链接的 `click` 事件上。

9.2 服务端管道

ASP.NET Web API 的服务端处理框架采用了管道式设计，这是一个不同于 ASP.NET MVC 的处理管道。笔者曾经写邮件咨询过 Scott Guthrie 为何不让 Web API 和 MVC 共享相同的管道，他回信说在设计 Web API 的时候确实想过重用 ASP.NET MVC 管道，但是很多实

现在 Web API 中请求处理功能却实现不了，所以才为 Web API 重建了另一个独立的管道。我个人觉得 ASP.NET Web API 管道从可扩展性上讲要优于 ASP.NET MVC 管道的设计，也许在下一个版本中微软会设计一个能够被 ASP.NET MVC 和 ASP.NET Web API 共用的管道。

我们通过整整 8 章的内容来介绍 ASP.NET MVC 管道对请求的处理，而 ASP.NET Web API 是一个独立的管道，所以我们不可能按照之前的方式对它进行如此详细的介绍。不过在 ASP.NET Web API 在进行请求处理的某些环节与 ASP.NET MVC 是很类似的，相信经过本章的介绍读者依然可以对 ASP.NET Web API 的运行机制有一个深刻的认识。

9.2.1 ASP.NET Web API 管道式设计

通过第 2 章“URL 路由”的介绍我们知道，URL 路由是一个独立于 ASP.NET MVC 的系统，它是进入 ASP.NET MVC 的入口。URL 路由对于 ASP.NET MVC 的主要作用在于通过注册的路由表对请求的 URL 进行解析，进而得到目标 Controller 和 Action 的名称以及其他相关的路由数据。

具体来说，针对 ASP.NET MVC 的路由表是一个 Route 对象的集合，每个 Route 对象关联着一个类型为 MvcHandler 的 IHttpHandler 对象。匹配成功的 Route 对象利用与之关联的 MvcHandler 对象来接手对当前请求的处理，而后者将请求递交给 ASP.NET MVC 框架进行后续处理。

ASP.NET Web API 的请求分发机制与之类似，两者的差异体现在采用的路由类型的不同。如图 9-4 所示，ASP.NET Web API 采用的路由对象类型为 HttpWebRoute（这是一个内部类型），而与之关联的 IHttpHandler 类型为 System.Web.Http.WebHost.HttpControllerHandler。我们可以将 HttpControllerHandler 视为 URL 路由系统与 ASP.NET Web API 之间的桥梁。

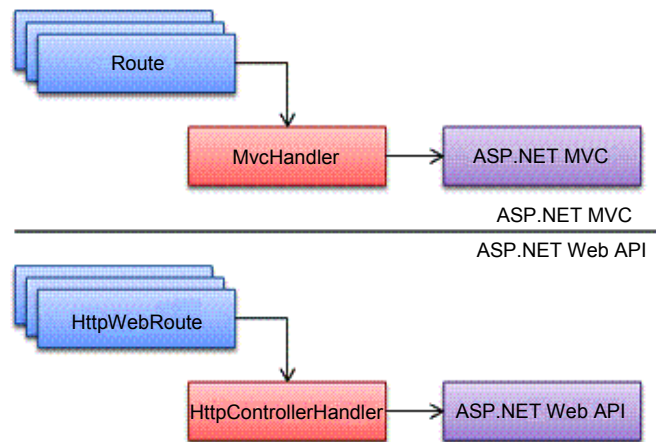


图 9-4 URL 路由系统与 ASP.NET MVC/ASP.NET Web API

如下所示的三个针对 `RouteCollection` 类型的扩展方法 `MapHttpRoute` 用于注册 `HttpWebRoute`。与 ASP.NET MVC 注册 `Route` 对象一样，我们可以指定注册项的名称、URL 模板、默认变量值和约束。

```
public static class RouteCollectionExtensions
{
    public static Route MapHttpRoute(this RouteCollection routes, string name,
        string routeTemplate);
    public static Route MapHttpRoute(this RouteCollection routes, string name,
        string routeTemplate, object defaults);
    public static Route MapHttpRoute(this RouteCollection routes, string name,
        string routeTemplate, object defaults, object constraints);
    public static Route MapHttpRoute(this RouteCollection routes, string name,
        string routeTemplate, object defaults, object constraints,
        HttpResponseMessage handler);
}
```

通过调用扩展方法 `MapHttpRoute` 注册的 `HttpWebRoute` 与一个类型为 `HttpControllerHandler` 的 `Handler` 进行映射。值得一提的是，ASP.NET Web API 采用了单例的模式来提供这个 `HttpControllerHandler` 对象，也就是说所有被注册的 `HttpWebRoute` 共享同一个 `HttpControllerHandler`。如下面的代码片段所示，`HttpControllerHandler` 实现了接口 `IHandler` 和 `IAsyncHandler`，所以默认情况下是以异步的方式（`BeginProcessRequest/EndProcessRequest`）执行的。`HttpControllerHandler` 是根据表示路由数据的 `RouteData` 对象创建的。

```
public class HttpControllerHandler : IAsyncHandler, IHandler
{
    //其他成员
    public HttpControllerHandler(RouteData routeData);

    IAsyncResult IAsyncHandler.BeginProcessRequest(HttpContext httpContext,
        AsyncCallback callback, object state);
    void IAsyncHandler.EndProcessRequest(IAsyncResult result);
    void IHandler.ProcessRequest(HttpContext httpContext);

    protected virtual bool IsReusable { get; }
    bool IHandler.IsReusable { get; }
}
```

ASP.NET Web API 服务端框架采用管道式设计，具有如图 9-5 所示结构的处理管道就是通过 `HttpControllerHandler` 创建的。当 `HttpControllerHandler` 的 `BeginProcessRequest` 方法被执行的时候，它会将请求分发给作为 ASP.NET Web API 管道入口的 `HttpServer` 对象，紧随 `HttpServer` 的是一系列 `HttpMessageHandler`，它们按照排列的顺序对从 `HttpServer` 传递过来的请求作进一步处理，处理后请求被最终转发给 `HttpControllerDispatcher`，由它激活 `HttpController` 并执行目标 `Action` 方法。

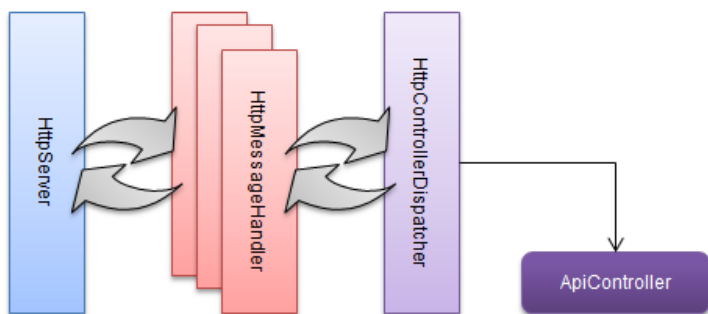


图 9-5 ASP.NET Web API 服务端管道

管道式的设计为 ASP.NET Web API 带来了很好的可扩展性，除了管道两端的 `HttpServer` 和 `HttpControllerDispatcher`，中间的 `HttpMessageHandler` 是可以定制的。如果我们需要对请求进行额外的处理，可以将相应的处理逻辑实现在自定义的 `HttpMessageHandler` 中，最终通过注册使它成为整个管道的某个“环节”参与对请求的处理。接下来我们就来详细地介绍这个主要由 `HttpServer`、`HttpMessageHandler` 和 `HttpControllerDispatcher` 组成的服务端管道。

9.2.2 `HttpMessageHandler`

虽然 ASP.NET Web API 由管道由 `HttpServer`、`HttpMessageHandler` 和 `HttpControllerDispatcher` 构成，但是 `HttpServer` 和 `HttpControllerDispatcher` 可以看成是两种特殊的 `HttpMessageHandler`，所以整个管道可以视为一个 `HttpMessageHandler` 链。`HttpMessageHandler` 继承自具有如下定义的抽象类型 `System.Net.Http.HttpMessageHandler`。

```
public abstract class HttpMessageHandler : IDisposable
{
    protected abstract Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken);

    public void Dispose();
    protected virtual void Dispose(bool disposing);
}
```

`HttpMessageHandler` 是一个抽象类，抽象方法 `SendAsync` 用于处理分发给它的请求。`HttpMessageHandler` 采用异步方式进行请求的处理，这可以从 `SendAsync` 方法的命名和返回类型（`Task<T>`）体现出来。随着对 ASP.NET Web API 管道的进一步深入了解，我们会发现几乎所有组件的 API 都采用这样的定义方式。

作为 `SendAsync` 方法参数的 `System.Net.Http.HttpRequestMessage` 对象是对当前请求的封装，ASP.NET Web API 还定义了另一个与之对应的类型 `System.Net.Http.HttpResponseMessage` 实现对响应的封装。

图 9-6 基本上体现了作为服务端管道的 `HttpMessageHandler` 链针对请求和响应的处理流程。每个 `HttpMessageHandler` 实现了 `ProcessRequest` 和 `ProcessResponse` 这两个基本操作，分

别进行请求和响应的处理。针对请求的处理实现在 `SendAsync` 方法中,而针对响应的处理则通过返回的 `Task<HttpResponseBody>` 对象来完成。分别封装了请求和响应的 `HttpRequestMessage` 和 `HttpResponseMessage` 可以视为管道处理过程中的上下文。换句话说,所有 `HttpMessageHandler` 处理都是相同的 `HttpRequestMessage` 和 `HttpResponseMessage` 对象。

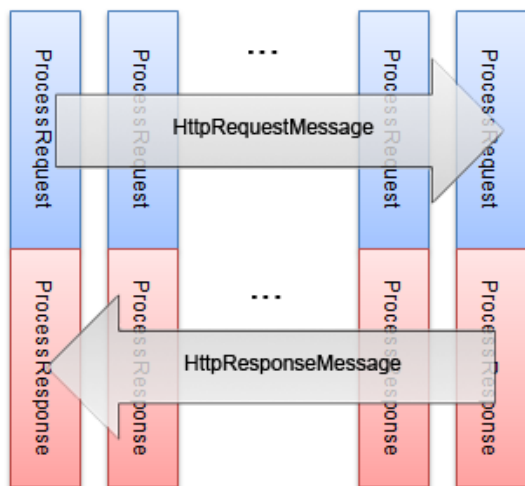


图 9-6 `HttpMessageHandler` 管道对请求/响应的处理

`HttpRequestMessage` 与 `HttpResponseMessage`

无论是 ASP.NET MVC 还是 ASP.NET Web API,本质上就是一个“处理请求,回复响应”的管道,我们可以直接通过当前 `HttpContext` 获取表示请求和响应的 `HttpRequest` 和 `HttpResponse` 对象。ASP.NET Web API 定义的 `HttpRequestMessage` 和 `HttpResponseMessage` 可以分别视为对它们的封装。

具有如下定义的 `HttpRequestMessage` 封装了 `HttpRequest` 的基本属性,包括通过 `RequestUri` 和 `Method` 属性表示的请求地址和 HTTP 方法,以及通过 `Headers` 属性表示的 HTTP 报头列表。字典类型的属性 `Properties` 是附加到 `HttpRequestMessage` 对象的属性列表,而 `Version` 属性代表 HTTP 消息的版本(默认版本为 1.1)。

```
public class HttpRequestMessage : IDisposable
{
    //其他成员
    public Uri RequestUri { get; set; }
    public HttpMethod Method { get; set; }
    public HttpRequestHeaders Headers { get; }
    public IDictionary<string, object> Properties { get; }
    public Version Version { get; set; }
    public HttpContent Content { get; set; }
}
```

`HttpRequestMessage` 具有一个 `Content` 属性封装了与 HTTP 消息主体内容相关的信息,

其类型为 `System.Net.Http.HttpContent`。如下面的代码片段所示, `HttpContent` 是一个抽象类, 它定义了 `CopyToAsync` 和 `ReadAsByteArrayAsync` 两组方法实现针对主体内容的读写。`HttpContent` 的 `Headers` 属性返回一个 `System.Net.Http.Headers.HttpContentHeaders` 对象, 我们通过它得到请求/响应消息中与主体内容相关的 HTTP 报头, 比如表示编码和长度的“Content-Encoding”和“Content-Length”报头等。

```
public abstract class HttpContent : IDisposable
{
    //其他成员
    public Task<byte[]> ReadAsByteArrayAsync();
    public Task<Stream> ReadAsStreamAsync();
    public Task<string> ReadAsStringAsync();

    public Task CopyToAsync(Stream stream);
    public Task CopyToAsync(Stream stream, TransportContext context);

    public HttpContentHeaders Headers { get; }
}
```

HTTP 响应的基本信息被封装到具有如下定义的 `HttpResponseMessage` 类型中。它的 `RequestMessage` 返回一个 `HttpRequestMessage` 对象, 表示与之匹配的请求。属性 `StatusCode` 和 `ReasonPhrase` 分别表示响应状态码和相关的描述。`IsSuccessStatusCode` 属性用于判断当前是否是一个成功的响应, 所谓成功的响应指的是状态码在范围[200, 299]以内的响应。属性 `Headers` 表示响应的 HTTP 报头列表, 对应的类型为 `System.Net.Http.Headers.HttpResponseHeaders`。`Version` 属性依然表示 HTTP 消息的版本, 而响应消息主体内容的读写, 以及与主体内容相关的报头的获取可以通过属性 `Content` 表示的 `HttpContent` 来实现。

```
public class HttpResponseMessage : IDisposable
{
    //其他成员
    public HttpRequestMessage RequestMessage { get; set; }

    public HttpStatusCode StatusCode { get; set; }
    public string ReasonPhrase { get; set; }
    public bool IsSuccessStatusCode { get; }
    public HttpResponseMessageHeaders Headers { get; }
    public Version Version { get; set; }
    public HttpContent Content { get; set; }
}
```

DelegatingHandler

ASP.NET Web API 管道中用于处理请求和响应的 `HttpMessageHandler` 并不是孤立的, 而是作为一个完整的 `HttpMessageHandler` 链中的某个环节而存在的, 这种链式结构可以通过具有如下定义的 `System.Net.Http.DelegatingHandler` 来构建。

```
public abstract class DelegatingHandler : HttpMessageHandler
{
    //其他成员
    protected DelegatingHandler();
}
```

```
protected DelegatingHandler(HttpMessageHandler innerHandler);
protected override Task<HttpResponseMessage> SendAsync(
    HttpRequestMessage request, CancellationToken cancellationToken);

public HttpMessageHandler InnerHandler { get; set; }
}
```

如上面的代码片段所示, `DelegatingHandler` 是一个继承自 `HttpMessageHandler` 的抽象类。它的属性 `InnerHandler` 表示它内部引用的另一个 `HttpMessageHandler`。 `DelegatingHandler` 实现了 `SendAsync` 方法, 但是在该方法中它什么都没有做, 仅仅简单地调用了这个 `InnerHandler` 的同名方法而已。这也是为何将其命名为 `DelegatingHandler` 的原因, 因为它仅仅是“委托”另一个 `HttpMessageHandler` 去真正完成对请求和响应的处理。

如果从数据结构的角度来讲, `DelegatingHandler` 采用了链表设计。如果我们按照如图 9-7 所示的方式将它的 `InnerHandler` 作为对链表中下一个 `DelegatingHandler` 的引用, 我们很容易将一组 `DelegatingHandler` 构建成一个 `HttpMessageHandler` 链, ASP.NET Web API 的服务端管道就是这么一个 `HttpMessageHandler` 链。

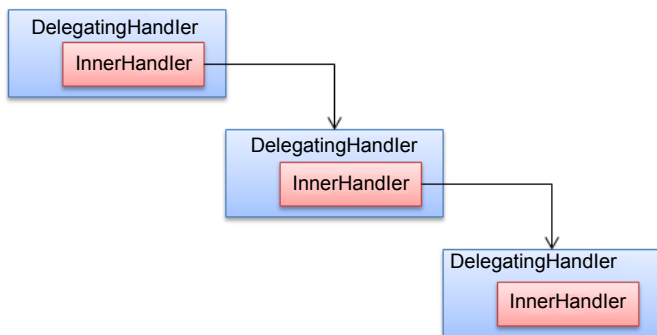


图 9-7 DelegatingHandler 管道

9.2.3 HttpServer

ASP.NET Web API 管道的第一个 `HttpMessageHandler` 是一个 `System.Web.Http.HttpServer` 对象。如下面的代码片段所示, `HttpServer` 实际上是 `DelegatingHandler` 的子类。它的只读属性 `Dispatcher` 返回作为分发器的 `HttpMessageHandler`, 这个分发器处于 `HttpMessageHandler` 链的尾端。如果在调用构造函数的时候没有通过参数显式指定, ASP.NET Web API 会创建一个 `HttpControllerDispatcher` 对象作为其默认分发器。

```
public class HttpServer : DelegatingHandler
{
    public HttpServer();
    public HttpServer(HttpControllerDispatcher dispatcher);
    public HttpServer(HttpConfiguration configuration);
    public HttpServer(HttpConfiguration configuration,
```

```
        HttpResponseMessage dispatcher);

    protected virtual void Initialize();
    protected override Task<HttpResponseBody> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken);

    public IConfiguration Configuration { get; }
    public HttpResponseMessage Dispatcher { get; }
}
```

HttpServer 的 Dispatcher 属性返回的 HttpResponseMessage 和从基类 DelegatingHandler 继承的 InnerHandler 属性返回的 HttpResponseMessage 是两个不同的对象，实际上如果我们直接调用构造函数创建一个 HttpServer 对象，它的 InnerHandler 属性为 Null。

HttpServer 另一个只读属性 Configuration 返回一个 System.Web.Http.HttpConfiguration 对象，它包含了控制 ASP.NET Web API 管道执行流程的配置信息。如果我们需要对 ASP.NET Web API 管道进行定制，在很多情况下都需要使用到它。如果构造函数中没有通过参数显式指定，会自动创建一个 HttpConfiguration 对象（调用默认构造函数）作为该属性的值。

HttpConfiguration

由于 HttpConfiguration 封装了用于控制整个 ASP.NET Web API 管道执行流程和行为的配置信息，我们很有必要了解一下它具体包含哪些配置，相应的配置控制着怎样的请求/响应处理行为。

```
public class HttpConfiguration : IDisposable
{
    //其他成员
    public IDependencyResolver DependencyResolver { get; set; }
    public HttpFilterCollection Filters { get; }
    public MediaTypeFormatterCollection Formatters { get; }
    public IncludeErrorDetailPolicy IncludeErrorDetailPolicy { get; set; }
    public Collection<DelegatingHandler> MessageHandlers { get; }
    public ServicesContainer Services { get; }

    public ConcurrentDictionary<object, object> Properties { get; }
}
```

上面的代码片段列出了 HttpConfiguration 的主要的属性成员，它们表示的含义以及针对请求/响应处理过程中的作用定义在下面的表格（表 9-1）中。

表 9-1 HttpConfiguration 常用属性

属 性	描 述
DependencyResolver	该属性的类型为 System.Web.Http.Dependencies.IDependencyResolver，用于实现基于依赖注入的组件提供/激活机制，它与 ASP.NET MVC 下的 DependencyResolver（对应的类型实现了 System.Web.Mvc.IDependencyResolver 接口）具有相同的作用。我们将在“HttpController 的激活”部分对它进行单独介绍

续表

属 性	描 述
Filters	该属性代表全局筛选器列表。与 ASP.NET MVC 类似，基于筛选器的横切关注点注入机制(AOP)可以通过筛选器应用到 ASP.NET Web API 上，ASP.NET Web API 为 AuthorizationFilter、ActionFilter 和 ExceptionFilter（没有 ResultFilter）这三种类型的筛选器定义了相应的接口，我们将在“Action 的执行与结果的响应”中对它们进行单独介绍
Formatters	该属性返回的是一个元素类型为 System.Net.Http.Formatting.MediaTypeFormatter 的列表。MediaTypeFormatter 是基于某种媒体类型的消息格式化器，这个列表实现了针对请求消息和响应消息的格式化（序列化），我们将在“Action 的执行与结果的响应”中对 MediaTypeFormatter 进行单独介绍
IncludeErrorDetailPolicy	该属性与异常处理策略相关，表示抛出异常的详细信息是否会包含在错误消息中返回。出于对安全的考虑，该属性默认值为 False，在开发调试阶段可以通过设置该属性使客户端直接得到异常的详细信息
Properties	该属性返回一个字典对象，所有的字典元素将会自动附加到表示当前请求的 HttpRequestMessage 上，我们可以通过 HttpRequestMessage 的 Properties 属性得到它们。
Services	<p>该属性返回一个 System.Web.Http.Services.ServicesContainer 对象，我们可以将它视为一个简易的 IoC 容器。ServicesContainer 维护一个服务实例和服务类型之间的匹配关系（这里的服务指的是 ASP.NET Web API 管道中某个提供某种功能并且可被替换的组件，其类型一般实现某个接口或者继承某个抽象类）。我们可以通过 ServicesContainer 将一个具体的服务实例注册到某个接口/抽象类上，在进行服务消费的时候就能够根据接口/抽象类将注册的服务实例提取出来。</p> <p>ASP.NET Web API 在进行请求/响应处理过程中的很多功能都是通过注册的服务来完成的，如果注册的默认服务不能满足我们的需要，可以通过实现对应的接口或者继承（直接或者间接）相应抽象类创建自定义的服务类型，并在应用启动的时候注册新的服务实例来替换默认注册的服务实例。</p> <p>ServicesContainer 是一个抽象类型，ASP.NET Web API 默认使用的是继承它的类型 System.Web.Http.Services.DefaultServices</p>

在默认的情况下，ASP.NET Web API 用于控制整个管道执行流程的 HttpConfiguration 是通过静态类型 System.Web.Http.GlobalConfiguration 定义的，这个 HttpConfiguration 对应着具有如下定义的静态只读属性 Configuration。GlobalConfiguration 还定义了另一个静态只读属

性 Dispatcher 返回默认作为分发器的 `HttpControllerDispatcher` 对象。

```
public static class GlobalConfiguration
{
    public static IConfiguration Configuration { get; }
    public static HttpControllerDispatcher Dispatcher { get; }
}
```

HttpServer 对请求的处理

作为 ASP.NET Web API 服务端管道“龙头”的 `HttpServer` 是通过 `HttpControllerHandler` 创建的。具体来说，`HttpControllerHandler` 采用延迟加载的方式来创建 `HttpServer`，`GlobalConfiguration` 的静态属性 `Configuration` 和 `Dispatcher` 表示的 `IConfiguration` 和 `HttpControllerDispatcher` 对象作为调用的构造函数参数。

当 `HttpControllerHandler` 的 `BeginProcessRequest` 方法被执行的时候，它会根据当前请求创建一个 `HttpRequestMessage` 对象。具体来说，它会根据当前请求的 HTTP 方法和 URL 创建一个 `HttpRequestMessage` 对象，并将当前 HTTP 报头拷贝到通过 `Headers` 属性表示的报头集合中。作为 `Content` 属性的 `HttpContent` 是一个根据当前请求的输入流（对应 `HttpRequest` 的 `InputStream` 属性）创建的 `System.Net.Http.StreamContent` 对象。

`HttpControllerHandler` 还会将当前的 `HttpContext` 和路由数据添加到 `HttpRequestMessage` 的 `Properties` 字典中，对应的 Key 分别为“MS_HttpContext”和“MS_HttpRouteData”，我们可以调用 `HttpRequestMessage` 具有如下定义的扩展方法 `GetRouteData` 得到这个保存的路由数据。

```
public static class HttpRequestMessageExtensions
{
    //其他成员
    public static IHttpRouteData GetRouteData(this HttpRequestMessage request);
}
```

`HttpControllerHandler` 创建的 `HttpRequestMessage` 接下来被传入管道进行处理。具体来说，它会将 `HttpRequestMessage` 对象作为参数调用 `HttpServer` 的 `SendAsync` 方法，并将返回的 `Task<HttpResponseMessage>` 对象封装成 `AsyncResult` 对象并以异步的形式执行。

通过前面的介绍我们知道，直接通过调用构造函数创建的 `HttpServer` 通过 `Dispatcher` 属性引用着作为分发器的 `HttpControllerDispatcher` 对象，但是此时通过它的 `InnerHandler` 属性表示的“下一个 `HttpMessageHandler`”尚未初始化，换句话说，真正的 `HttpMessageHandler` 管道此时尚未构建。在它的 `SendAsync` 方法被第 1 次调用之前具有一个 `HttpMessageHandler` 管道初始化的过程，在这个过程中，添加到 `HttpConfiguration` 的 `MessageHandlers` 属性中的每个 `HttpMessageHandler` 按照先后顺序被提取出来，最终构建出如图 9-5 所示的 `HttpMessageHandler` 管道。

我们可以通过一个简单的实例来演示 `HttpServer` 初始化过程中对整个 ASP.NET Web API 管道的构建，我们在一个 ASP.NET MVC 应用中（不是 ASP.NET Web API 应用）创建了如下三个继承自 `DelegatingHandler` 的自定义 `HttpMessageHandler`。

```

public class FooMessageHandler : DelegatingHandler
{ }

public class BarMessageHandler : DelegatingHandler
{ }

public class BazMessageHandler : DelegatingHandler
{ }

```

然后我们定义了如下一个 `HomeController`，辅助方法 `GetPipeline` 返回一个字符串用以描述组成 `HttpMessageHandler` 管道的每个 `HttpMessageHandler` 对象的先后顺序，而作为参数传入的是作为管道的初始端的 `HttpMessageHandler` 对象。我们在 `Action` 方法 `Index` 中创建一个 `HttpConfiguration` 对象，并将创建的三个自定义 `HttpMessageHandler` 添加到它的 `MessageHandlers` 集合中，最后基于这个 `HttpConfiguration` 创建一个 `HttpServer` 对象。接下来在调用其 `SendAsync` 方法（由于这是一个内部方法，我们你不得不采用反射的方式进行调用）的前后分别调用 `GetPipeline` 方法将由它“牵头”的 `HttpMessageHandler` 管道呈现出来。

```

public class HomeController : Controller
{
    public void Index()
    {
        HttpConfiguration configuration = new HttpConfiguration();
        configuration.MessageHandlers.Add(new FooMessageHandler());
        configuration.MessageHandlers.Add(new BarMessageHandler());
        configuration.MessageHandlers.Add(new BazMessageHandler());

        HttpServer httpServer = new HttpServer(configuration);
        Response.Write("初始化前: " + GetPipeline(httpServer) + "<br/>");

        //以反射的方式执行 SendAsync 方法促使 HttpMessageHandler 管道的创建
        MethodInfo method = typeof(HttpMessageHandler).GetMethod("SendAsync",
            BindingFlags.Instance | BindingFlags.NonPublic);
        method.Invoke(httpServer, new object[] { new HttpRequestMessage(),
            new CancellationToken() });
        Response.Write("初始化后: " + GetPipeline(httpServer) + "<br/>");
    }

    private string GetPipeline(HttpMessageHandler httpServer)
    {
        string pipeline = httpServer.GetType().Name;
        DelegatingHandler delegatingHandler = httpServer as DelegatingHandler;
        if (null != delegatingHandler && delegatingHandler.InnerHandler != null)
        {
            return pipeline + " => " +
                GetPipeline(delegatingHandler.InnerHandler);
        }
        else
        {
            return pipeline;
        }
    }
}

```

运行该程序的时候会在浏览器中会呈现出如图 9-8 所示的输出结果，可以清楚地看到当

调用构造函数创建 `HttpServer` 的时候，它仅仅是一个孤立的对象。在我们成功调用了它的 `SendAsync` 方法后，通过 `HttpConfiguration` 注册的 `HttpMessageHandler` 会连接成串，而它们首尾连接的正是 `HttpServer` 和作为分发器的 `HttpControllerDispatcher` 对象。（S902）

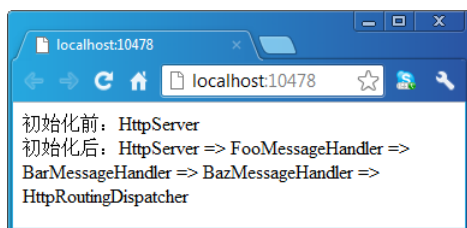


图 9-8 `HttpServer` 的 `SendAsync` 方法的初次调用导致了 `HttpMessageHandler` 管道的构建

当整个 `HttpMessageHandler` 管道成功构建之后，如果通过 `SynchronizationContext` 的静态属性 `Current` 表示的同步上下文存在的话，它将作为消息属性附加到表 `HttpRequestMessage` 的 `Properties` 属性上，对应的 Key 为 “MS_SynchronizationContext”。除此之外，作为消息属性一并被保存的还包括当前的 `HttpConfiguration` 对象，对应的 Key 为 “MS_HttpConfiguration”。

如果我们需要使用 `SynchronizationContext` 进行一些同步控制，或者需要利用 `HttpConfiguration` 设置或者获取相应的配置，可以通过相应的 Key 从 `HttpRequestMessage` 对象的 `Properties` 属性中提取这两个对象，不过更好的方式还是直接通过具有如下定义的两个扩展方法来得到它们。

```
public static class HttpRequestMessageExtensions
{
    // 其他成员
    public static HttpConfiguration GetConfiguration(
        this HttpRequestMessage request);
    public static SynchronizationContext GetSynchronizationContext(
        this HttpRequestMessage request);
}
```

9.2.4 实例演示：自定义 `HttpMessageHandler` 实现 HTTP 方法重写（S903）

一个 RESTful 服务一般都采用 URI 表示操作的资源，用 HTTP 方法（PUT、GET、POST 和 DELETE 等）表示针对资源的 CRUD 操作，但是一些客户端只支持 GET 和 POST 这两种基本的 HTTP 方法，这意味着它不能正常发送 PUT 和 DELETE 请求对资源作添加和删除操作。在一般情况下我们利用“HTTP 方法重写”的方式来解决这个问题。

具体来说，针对资源添加和删除操作的 HTTP 请求依然可以采用 POST 方法，但是它会为请求添加一个名为“X-HTTP-Method-Override”的报头表示希望被重写的 HTTP 方法（PUT

或者 DELETE)，接收该请求的服务端根据这个报头的值而非真正的 HTTP 方法来处理请求。

ASP.NET Web API 采用管道式的设计，这使我们可以很容易地通过自定义 `HttpMessageHandler` 实现基于“X-HTTP-Method-Override”报头的重写，因为 `HttpRequestMessage` 表示 HTTP 方法的 `Method` 属性是可读可写的，我们只需要利用自定义的 `HttpMessageHandler` 将“X-HTTP-Method-Override”报头的值赋给该属性即可。

为此我们定义了如下一个继承自 `DelegatingHandler` 的 `HttpMethodOverrideHandler`，在重写的 `SendAsync` 方法中实现了对“X-HTTP-Method-Override”报头的提取和对当前 HTTP 方法的重写。

```
public class HttpMethodOverrideHandler: DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationTokn)
    {
        IEnumerable<string> methodOverrideHeader;
        if (request.Method == HttpMethod.Post && request.Headers.TryGetValues(
            "X-HTTP-Method-Override", out methodOverrideHeader))
        {
            request.Method = new HttpMethod(methodOverrideHeader.First());
        }
        return base.SendAsync(request, cancellationTokn);
    }
}
```

我们将自定义的 `HttpMethodOverrideHandler` 应用到最初我们创建的用于进行联系人管理的 Web API 上，为此在 `Global.asax` 中通过如下的代码对 `HttpMethodOverrideHandler` 进行注册。

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        GlobalConfiguration.Configuration.MessageHandlers.Add(
            new HttpMethodOverrideHandler());
    }
}
```

创建一个控制台应用来作为调用 Web API 的客户端程序。简单起见，我们直接利用 `System.Net.Http.HttpClient` 来进行 Web API 的调用（我们需要为控制台应用添加该类型所在程序集 `System.Net.Http.dll` 的引用）。如下面的代码片段所示，我们为创建的 `HttpClient` 添加了一个值为“DELETE”的“X-HTTP-Method-Override”报头，并执行 `PostAsync` 方法调用 Web API 以删除 ID 号为“001”的联系人，为了确认联系人是否被成功删除，在调用前后获取并输出了整个联系人列表。

```
HttpClient httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Add("X-HTTP-Method-Override", "DELETE");
Task.WaitAll(httpClient.GetAsync("http://localhost:3721/api/contacts")
    .ContinueWith(response => Console.WriteLine(
```

```

        "删除前: \n{0}\n", response.Result.Content.ReadAsStringAsync().Result));
Task.WaitAll(httpClient.PostAsync("http://localhost:3721/api/contacts/001",
    null));

httpClient.GetAsync("http://localhost:3721/api/contacts")
    .ContinueWith(response => Console.WriteLine(
        "删除后: \n{0}\n", response.Result.Content.ReadAsStringAsync().Result));
Console.Read();

```

我们在启动承载 Web API 的应用程序（为了使客户端能够采用固定的 URL 进行服务调用，我们为 Visual Studio Development Server 设置了固定的端口 3721）后运行客户端程序，会在控制台上呈现出如下的输出结果，可以清楚地看到由于服务端利用了 `HttpMethodOverrideHandler` 实现的 HTTP 方法重写功能，客户端可以通过 POST 请求调用基于 DELETE 请求的服务操作。

删除前:

```

[{"Id": "001", "Name": "张三",
  "PhoneNo": "123", "EmailAddress": "zhangsan@gmail.com"},
 {"Id": "002", "Name": "李四",
  "PhoneNo": "456", "EmailAddress": "lisi@gmail.com"}]

```

删除后:

```

[{"Id": "002", "Name": "李四",
  "PhoneNo": "456", "EmailAddress": "lisi@gmail.com"}]

```

9.3 ApiControllerDispatcher

当 `ApiControllerHandler` 将根据当前请求创建的 `HttpRequestMessage` 递交给服务端管道作进一步处理时，位于管道首端的 `HttpServer` 会率先对其进行处理，随后 `HttpMessageHandler` 链从 `HttpServer` 接管 `HttpRequestMessage` 并作进一步处理。被处理后的 `HttpRequestMessage` 最终递交给位于管道尾端的 `ApiControllerDispatcher`，由它负责对请求作最后的处理，而这个过程就包括对 `ApiController` 的激活和对目标 `Action` 的执行。

```

public class ApiControllerDispatcher : HttpMessageHandler
{
    public ApiControllerDispatcher();
    public ApiControllerDispatcher(HttpConfiguration configuration);
    protected override void Dispose(bool disposing);
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken);

    public HttpConfiguration Configuration { get; }
}

```

如上面的代码片段所示，继承自 `HttpMessageHandler` 的 `ApiControllerDispatcher` 同样具有一个 `HttpConfiguration` 的只读属性 `Configuration`，该属性在构造函数中初始化。`ApiControllerDispatcher` 是整个服务端管道的核心，在这一节中我们会详细介绍实现在 `ApiControllerDispatcher` 中针对请求/响应的处理流程。

9.3.1 ApiController 的激活

通过前面的介绍我们知道，`ApiControllerHandler` 在根据当前请求创建出 `HttpRequestMessage` 后会将路由数据添加到该对象的 `Properties` 属性表示的字典中。`ApiControllerDispatcher` 在接收到 `HttpRequestMessage` 后的第一项工作就是提取路由数据，并根据目标的 Controller 名称激活对应的 `ApiController` 对象。不过在正式介绍 `ApiController` 的激活之前，我们需要了解一下用于描述 `ApiController` 的类型 `System.Web.Http.Controllers.ApiControllerDescriptor`。

ApiControllerDescriptor

ASP.NET Web API 下的 `ApiControllerDescriptor` 相当于 ASP.NET MVC 下的 `ControllerDescriptor`，是对 `ApiController` 的描述。如下面的代码片段所示，`ApiControllerDescriptor` 具有三个可读写属性 `ControllerName`、`ControllerType` 和 `Configuration`，它们分别表示 `ApiController` 的名称、类型和对应的 `HttpConfiguration`，这三个属性可以通过构造函数的方式进行初始化。

```
public class ApiControllerDescriptor
{
    public ApiControllerDescriptor();
    public ApiControllerDescriptor(HttpConfiguration configuration,
        string controllerName, Type controllerType);

    public virtual IHttpController CreateController(HttpRequestMessage request);
    public virtual Collection<T> GetCustomAttributes<T>() where T: class;
    public virtual Collection<IFilter> GetFilters();

    public string ControllerName { get; set; }
    public Type ControllerType { get; set; }
    public HttpConfiguration Configuration { get; set; }

    public IActionValueBinder ActionValueBinder { get; set; }
    public IHttpActionInvoker HttpActionInvoker { get; set; }
    public IHttpActionSelector HttpActionSelector { get; set; }

    public IHttpControllerActivator ApiControllerActivator { get; set; }
    public ConcurrentDictionary<object, object> Properties { get; }
}
```

属性 `ActionValueBinder` 的作用类似于 ASP.NET MVC 中的 `ModelBinder`，用于绑定 Action 方法的输入参数。属性 `HttpActionInvoker` 则相当于 ASP.NET MVC 下的 `ActionInvoker`，用于实现最终对 Action 的调用。`HttpActionSelector` 属性返回用于获取当前 Action 的选择器。

ASP.NET MVC 通过 `ControllerActivator` 激活目标 Controller，而 ASP.NET Web API 则利用 `ApiControllerActivator` 来激活 `ApiController`，`ApiControllerActivator` 属性返回的就是这么一个对象。`ApiController` 的激活可以直接调用 `CreateController` 方法来完成，该方法在内部还是借助于 `ApiControllerActivator` 来创建相应的 `ApiController` 对象。

方法 `GetCustomAttributes<T>` 帮助我们获取应用在 `HttpController` 类型上指定类型的特性，而另一个方法 `GetFilters` 则用于获取应用在 `HttpController` 上的所有筛选器。除此之外，`HttpControllerDescriptor` 同样具有一个字典类型的 `Properties` 属性保存与之关联的属性。

HttpController 类型解析

`HttpController` 能够被成功激活的前提是我们能够正确解析出它对应的类型，ASP.NET Web API 采用了与 ASP.NET MVC 相同的 `Controller` 类型解析机制，它将所有实现了 `IHttpController` 接口的类型提取出来，然后通过路由数据中的 `Controller` 名称去匹配。

ASP.NET Web API 定义了如下一个 `System.Web.Http.Dispatcher.IHttpControllerTypeResolver` 接口，其 `GetControllerTypes` 方法用于获取所有 `HttpController` 类型列表。该方法具有一个类型为 `System.Web.Http.Dispatcher.IAssembliesResolver` 的参数 `assembliesResolver`，用于辅助完成针对程序集的解析。

```
public interface IHttpControllerTypeResolver
{
    ICollection<Type> GetControllerTypes(IAssembliesResolver assembliesResolver);
}

public interface IAssembliesResolver
{
    ICollection<Assembly> GetAssemblies();
}
```

`HttpControllerDispatcher` 是如何利用 `HttpControllerTypeResolver` 获取所有 `HttpController` 的类型列表的呢？这个过程涉及到了创建 `HttpControllerDispatcher` 时指定的 `HttpConfiguration` 对象。在前面介绍 `HttpConfiguration` 的时候，我们谈到它具有一个类型为 `ServicesContainer` 的 `Services` 属性，可以利用它将一个具体的服务对象注册到对应的服务接口上。

`HttpControllerDispatcher` 可以直接利用 `ServicesContainer` 根据 `IHttpControllerTypeResolver` 接口得到注册的 `HttpControllerTypeResolver` 对象。实际上我们可以直接调用如下两个针对 `ServicesContainer` 的扩展方法 `GetAssembliesResolver` 和 `GetHttpControllerTypeResolver` 来获取注册的 `AssembliesResolver` 和 `HttpControllerTypeResolver`。

```
public static class ServicesExtensions
{
    //其他成员
    public static IAssembliesResolver GetAssembliesResolver(
        this ServicesContainer services);
    public static IHttpControllerTypeResolver GetHttpControllerTypeResolver(
        this ServicesContainer services);
}
```

在默认情况下注册的 `AssembliesResolver` 和 `HttpControllerDispatcher` 类型是哪个好呢？从如下所示的 `DefaultServices` 构造函数的定义中不难看出，在默认情况下注册的 `Assemblies`

Resolver 类型为 `DefaultAssembliesResolver`, 而注册的 `HttpControllerTypeResolver` 则是一个类型为 `DefaultHttpControllerTypeResolver` 的对象。这里默认注册的两个类型 (`DefaultAssembliesResolver` 和 `DefaultHttpControllerTypeResolver`) 均为内部类型。

```
public class DefaultServices : ServicesContainer
{
    //其他成员
    public DefaultServices(HttpConfiguration configuration)
    {
        //其他操作
        this.SetSingle<IAssembliesResolver>(new DefaultAssembliesResolver());
        this.SetSingle<IHttpControllerTypeResolver>(
            new DefaultHttpControllerTypeResolver());
    }
}
```

但是 `HttpControllerDispatcher` 真正使用的 `AssembliesResolver` 和 `HttpControllerTypeResolver` 类型却不是 `DefaultAssembliesResolver` 和 `DefaultHttpControllerTypeResolver`。通过前面的介绍我们知道, `HttpControllerHandler` 创建 `HttpControllerDispatcher` 时指定的 `HttpConfiguration` 是通过 `GlobalConfiguration` 的静态属性 `Configuration` 获取的。如下面的代码片段所示, 当 `GlobalConfiguration` 的 `Configuration` 被使用的时候会对这两个服务进行重新注册。

```
public static class GlobalConfiguration
{
    //其他成员
    private static Lazy<HttpConfiguration> _configuration =
        new Lazy<HttpConfiguration>(delegate {
            //...
            configuration.Services.Replace(typeof(IAssembliesResolver),
                new WebHostAssembliesResolver());
            configuration.Services.Replace(typeof(IHttpControllerTypeResolver),
                new WebHostHttpControllerTypeResolver());
            return configuration;
        });
}
```

从上面的代码片段中不难看出, `HttpControllerDispatcher` 在默认情况下用于获取所有程序集和 `HttpController` 类型的 `AssembliesResolver` 和 `HttpControllerTypeResolver` 类型分别是 `WebHostAssembliesResolver` 和 `WebHostHttpControllerTypeResolver`。`WebHostAssembliesResolver` 直接调用 `BuildManager` 的静态方法 `GetReferencedAssemblies` 获取所有引用的程序集。而 `WebHostHttpControllerTypeResolver` 则先利用 `AssembliesResolver` 获取所有引用的程序集, 然后从中解析出所有实现了 `IHttpController` 接口的非抽象类型。

为了避免 `HttpController` 类型解析导致的频繁反射操作, `WebHostHttpControllerTypeResolver` 会对解析出来的 `HttpController` 类型进行缓存。具体的缓存策略与 ASP.NET MVC 解析 `Controller` 类型时一致, 即将类型列表进行序列化生成一个 XML 文件保存在 ASP.NET 的临时目录下。这个 XML 文件的文件名为 “MS-ApiControllerTypeCache.xml”, 我们可以在如下两个目录 (之一) 下得到这个文件。

- %Windir%\Microsoft.NET\Framework\v{version}\TemporaryASP.NET Files\{appname}\...\..\UserCache\
- %Windir%\Microsoft.NET\Framework\v{version}\Temporary ASP.NET Files\root\...\..\User Cache\

为了让读者对 `HttpController` 类型的解析具有一个深刻的认识，我们来演示一下如何获取所有 `HttpController` 类型列表。在一个 ASP.NET MVC 应用中定义如下三个继承自 `ApiController` 的 `HttpController`。

```
public class FooController : ApiController
{
}

public classBarController : ApiController
{
}

public class BazController : ApiController
{
}
```

接下来创建如下一个默认的 `HomeController`。在 `Action` 方法 `Index` 中，我们利用 `GlobalConfiguration` 获取当前注册的 `AssembliesResolver` 和 `HttpControllerTypeResolver` 对象，然后调用 `HttpControllerTypeResolver` 的 `GetControllerTypes` 方法得到所有的 `HttpController` 类型并作为 `Model` 呈现在默认的 `View` 中。而在这之前，我们需要将 `AssembliesResolver` 和 `HttpControllerTypeResolver` 对象保存到 `ViewBag` 中。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        IHttpControllerTypeResolver controllerTypeResolver = GlobalConfiguration
            .Configuration.Services.GetHttpControllerTypeResolver();
        IAssembliesResolver assembliesResolver = GlobalConfiguration
            .Configuration.Services.GetAssembliesResolver();

        ViewBag.ControllerTypeResolver = controllerTypeResolver;
        ViewBag.AssembliesResolver = assembliesResolver;

        return View(controllerTypeResolver.GetControllerTypes(
            assembliesResolver));
    }
}
```

如下所示的是 `Action` 方法 `Index` 对应 `View` 的定义，这是一个 `Model` 类型为 `IEnumerable<Type>` 的强类型 `View`。在该 `View` 中我们从 `ViewBag` 中将 `AssembliesResolver` 和 `HttpControllerTypeResolver` 对象提取出来，并将它们的类型连同作为 `Model` 的 `HttpController` 的类型列表呈现在一个表格中。

```
@model IEnumerable<Type>
<html>
    <head>
        <title>HttpController 类型的解析</title>
    </head>
```

```

<body>
  <table>
    <tr>
      <td>HttpControllerTypeResolver</td>
      <td>@ViewBag.ControllerTypeResolver.GetType().Name</td>
    </tr>
    <tr>
      <td>AssembliesResolver</td>
      <td>@ViewBag.AssembliesResolver.GetType().Name</td>
    </tr>
    @{
      var controllerTypes = Model.ToArray();
      <tr>
        <td rowspan="@Model.Count()">HttpController</td>
        <td>@controllerTypes[0].Name</td>
      </tr>
      for(int i=1; i<controllerTypes.Length; i++)
      {
        <tr><td>@controllerTypes[i].Name</td></tr>
      }
    }
  </table>
</body>
</html>

```

运行上面的程序会在开启的浏览器中呈现出如图 9-9 所示的输出结果。前面介绍的注册的 `AssembliesResolver` 和 `HttpControllerTypeResolver` 在这里得到了印证, 整个 Web 应用中定义的三个 `HttpController` 类型通过 `HttpControllerTypeResolver` 成功解析出来。(S904)

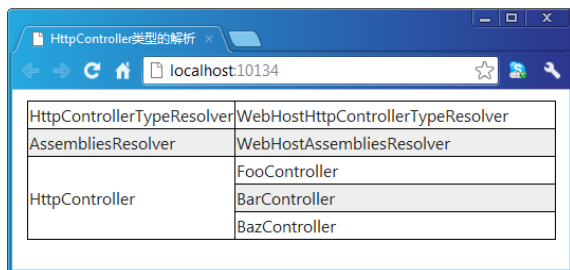


图 9-9 `HttpControllerTypeResolver` 对 `HttpController` 类型的解析

HttpController 的选择

ASP.NET Web API 利用 `HttpControllerTypeResolver` 解析出所有 `HttpController` 类型之后, 会根据保存在路由数据中的 `HttpController` 名称从中选择出当前请求的那一个, 目标 `HttpController` 的选择通过 `HttpControllerSelector` 来完成。 `HttpControllerSelector` 实现了具有如下定义的接口 `System.Web.Http.Dispatcher.IHttpControllerSelector`, 该接口定义了 `GetControllerMapping` 和 `SelectController` 两个方法, 前者返回用于描述所有 `HttpController` 的 `HttpControllerDescriptor` 对象与名称之间的映射关系, 后者从中选择出针对当前请求的 `HttpControllerDescriptor`。

```
public interface IHttpControllerSelector
{
    IDictionary<string, HttpControllerDescriptor> GetControllerMapping();
    HttpControllerDescriptor SelectController(HttpRequestMessage request);
}
```

HttpControllerSelector 依然注册在当前 HttpConfiguration 的 ServicesContainer 中, 可以调用 ServicesContainer 的扩展方法 GetHttpControllerSelector 得到它。默认注册的 IHttpControllerSelector 类型是一个具有如下定义的 System.Web.Http.Dispatcher.DefaultHttpControllerSelector。

```
public static class ServicesExtensions
{
    //其他成员
    public static IHttpControllerSelector GetHttpControllerSelector(
        this ServicesContainer services);
}

public class DefaultHttpControllerSelector : IHttpControllerSelector
{
    public DefaultHttpControllerSelector(HttpConfiguration configuration);

    public virtual IDictionary<string, HttpControllerDescriptor>
        GetControllerMapping();
    public virtual string GetControllerName(HttpRequestMessage request);
    public virtual HttpControllerDescriptor SelectController(
        HttpRequestMessage request);
}
```

DefaultHttpControllerSelector 直接利用 HttpControllerTypeResolver 得到所有 HttpController 的类型, 然后据此创建相应的 HttpControllerDescriptor 对象作为 GetControllerMapping 方法返回的字典对象的 Value, HttpController 类型名称去除“Controller”后缀得到的字符串作为返回字典对象的 Key。

对于实现的另一个方法 SelectController 来说, 它先从给定的 HttpRequestMessage 对象中获取目标 HttpController 的名称。具体的做法是: 采用预定义的 Key 从 HttpRequestMessage 的 Properties 属性中提取路由数据, 进而根据路由数据得到目标 HttpController 的名称, 最后根据此名称在通过 GetControllerMapping 方法获取的字典对象中获取匹配的 HttpControllerDescriptor 即可。根据 HttpRequestMessage 对 Controller 名称的解析实现在虚方法 GetControllerName 中。

我们同样通过一个简单的实例来演示如何通过 IHttpControllerSelector 实现对目标 HttpController 的选择。在一个 ASP.NET MVC 应用中定义了如下三个继承自 ApiController 的 HttpController 类型。

```
public class FooController : ApiController
{ }

public class BarController : ApiController
{ }
```



```
public class BazController : ApiController
{ }
```

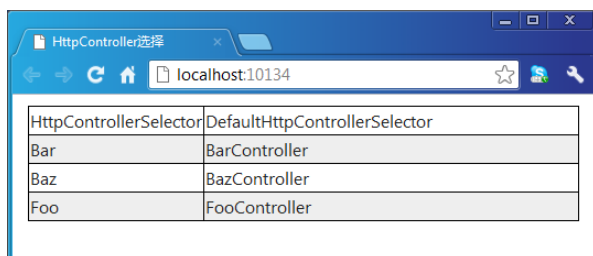
接下来定义如下一个 `HomeController`，在 `Action` 方法 `Index` 中我们通过静态类型 `GlobalConfiguration` 得到注册的 `HttpControllerSelector` 对象，并将调用其 `GetControllerMapping` 方法得到的字典对象作为 `Model` 呈现在默认的 `View` 中。而在这之前，我们将得到的 `HttpControllerSelector` 对象保存在 `ViewBag` 中。

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        IHttpControllerSelector controllerSelector = GlobalConfiguration
            .Configuration.Services.GetHttpControllerSelector();
        ViewBag.ControllerSelector = controllerSelector;
        return View(controllerSelector.GetControllerMapping());
    }
}
```

如下所示的是 `Action` 方法 `Index` 对应 `View` 的定义，这是一个 `Model` 类型为 `IDictionary<string, HttpControllerDescriptor>` 的强类型 `View`。在该 `View` 中我们遍历作为 `Model` 的字典对象，并利用 `HttpControllerDescriptor` 得到对应的 `HttpController` 的类型。我们将 `HttpController` 的名称(字典的 `Key`)和类型名称，以及从 `ViewBag` 提取的 `HttpControllerSelector` 对象的类型名称一并呈现在一个表格中。

```
@model IDictionary<string, HttpControllerDescriptor>
<html>
    <head>
        <title>HttpController 选择</title>
    </head>
    <body>
        <table>
            <tr>
                <td>HttpControllerSelector</td>
                <td>@ViewBag.ControllerSelector.GetType().Name</td>
            </tr>
            @foreach (var item in Model)
            {
                <tr>
                    <td>@item.Key</td><td>@item.Value.ControllerType.Name</td>
                </tr>
            }
        </table>
    </body>
</html>
```

上面这个程序运行之后会在浏览器中呈现出如图 9-10 所示的输出结果。默认注册的 `DefaultHttpControllerSelector` 在这里得到了印证，而通过调用它的 `GetControllerMapping` 方法确实能够得到用于描述所有 `HttpController` 的 `HttpControllerDescriptor` 对象和匹配的 `HttpController` 名称。(S905)



HttpControllerSelector	DefaultHttpControllerSelector
Bar	BarController
Baz	BazController
Foo	FooController

图 9-10 HttpControllerSelector 对 HttpController 的选择

HttpController 的创建

HttpControllerSelector 可以根据名称得到用于描述 HttpController 的 HttpControllerDescriptor 对象, 可以通过调用 HttpControllerDescriptor 对象的 CreateController 方法根据请求创建对应的 HttpController 对象。该方法在内部借助于一个叫做 HttpControllerActivator 的组件最终实现对目标 HttpController 的创建。HttpControllerActivator 实现了具有如下定义的接口 System.Web.Http.Dispatcher.IHttpControllerActivator, HttpController 的创建实现在 Create 方法中。

```
public interface IHttpControllerActivator
{
    IHttpController Create(HttpRequestMessage request,
        HttpControllerDescriptor controllerDescriptor, Type controllerType);
}
```

HttpControllerActivator 同样是注册在当前 HttpConfiguration 的 ServicesContainer 对象中, 我们可以通过 ServicesContainer 具有如下定义的扩展方法 GetHttpControllerActivator 得到注册的 HttpControllerActivator 服务实例。默认注册的 HttpControllerActivator 类型是具有如下定义的 System.Web.Http.Dispatcher.DefaultHttpControllerActivator。

```
public static class ServicesExtensions
{
    //其他成员
    public static IHttpControllerActivator GetHttpControllerActivator(
        this ServicesContainer services);
}

public class DefaultHttpControllerActivator : IHttpControllerActivator
{
    public DefaultHttpControllerActivator();
    public IHttpController Create(HttpRequestMessage request,
        HttpControllerDescriptor controllerDescriptor, Type controllerType);
}
```

DefaultHttpControllerActivator 在进行 HttpController 创建的时候利用了 DependencyResolver 实现了依赖注入机制。ASP.NET Web API 下的 DependencyResolver 对 ASP.NET MVC 下的 DependencyResolver 作了相应地改进, 我们可以将创建的服务实例限制在某个范围内以实现单独对它们进行释放。

```
public interface IDependencyResolver : IDependencyScope, IDisposable
{
    IDependencyScope BeginScope();
}

public interface IDependencyScope : IDisposable
{
    object GetService(Type serviceType);
    IEnumerable<object> GetServices(Type serviceType);
}
```

如上面的代码片段所示,真正用于获取服务实例的两个方法 `GetService` 和 `GetAllServices` 定义在 `System.Web.Http.Dependencies.IDependencyScope` 接口中。`IDependencyScope` 继承自 `IDisposable` 接口,对激活对象的释放应该定义在实现的 `Dispose` 方法中。`IDependencyResolver` 是 `IDependencyScope` 的继承者,它定义了一个 `BeginScope` 方法用于创建一个 `DependencyScope` 对象。

`HttpServer` 在进行请求处理的时候会将当前 `HttpConfiguration` 作为消息属性添加到 `HttpRequestMessage` 的 `Properties` 属性上(对应的 `Key` 为“`MS_HttpConfiguration`”),可以调用 `HttpRequestMessage` 具有如下定义的扩展方法 `GetConfiguration` 得到这个 `HttpConfiguration` 对象。

```
public static class HttpRequestMessageExtensions
{
    //其他成员
    public static HttpConfiguration GetConfiguration(
        this HttpRequestMessage request);
}
```

当 `DefaultHttpControllerActivator` 在进行 `HttpController` 激活的时候,会从当前请求中获取该 `HttpConfiguration` 对象,并通过它的 `DependencyResolver` 属性得到当前的 `DependencyResolver` 对象。具体来说,它会先调用 `DependencyResolver` 的 `BeginScope` 方法创建一个 `DependencyScope` 对象,然后进一步将 `HttpController` 的类型作为参数调用 `DependencyScope` 对象的 `GetService` 方法,如果返回值不为 `Null`,则直接作为激活的 `HttpController` 对象返回,否则直接对 `HttpController` 的类型实施反射以创建最终的 `HttpController` 对象。通过 `DependencyResolver` 创建的 `DependencyScope` 对象会在请求处理结束后被释放。

默认情况下注册的 `DependencyResolver` 对象类型是一个名称为 `EmptyResolver` 的内部类型,实际上它并不做任何的对象创建工作,其 `GetService` 直接返回 `Null`,所以默认情况下激活的 `HttpController` 最终还是通过对类型反射来创建的。

实例演示: 通过自定义 `HttpControllerActivator` 实现基于 IoC 的 `HttpController` 的激活 (S906)

将 IoC 集成到 `HttpController` 激活过程中具有重要的现实意义。在第3章“Controller 的激活”中我们通过自定义 `ControllerFactory`、`ControllerActivator` 和 `DependencyResolver` 的方式实现了基于 IoC 的 `Controller` 激活方式,现在我们采用类似的方式将 IoC 引入 ASP.NET Web API 中实现针对 `HttpController` 的激活。

我们通过实例来演示如何通过自定义 `HttpControllerActivator` 的方式实现与 IoC 的集成。采用的 IoC 框架是 Unity，为此我们定义了名为 `UnityHttpControllerActivator` 的自定义 `HttpControllerActivator`。如下面的代码片段所示，`UnityHttpControllerActivator` 具有一个表示 Unity 容器的只读属性 `UnityContainer`，该属性在构造函数中被初始化，在用于创建 `HttpController` 的 `Create` 方法中，我们将 `HttpController` 的类型作为参数调用 `UnityContainer` 的 `Resolve` 方法并将返回的对象作为激活的 `HttpController`。

```
public class UnityHttpControllerActivator : IHttpControllerActivator
{
    public IUnityContainer UnityContainer { get; private set; }

    public UnityHttpControllerActivator(IUnityContainer unityContainer)
    {
        this.UnityContainer = unityContainer;
    }

    public IHttpController Create(HttpRequestMessage request,
        HttpControllerDescriptor controllerDescriptor, Type controllerType)
    {
        return (IHttpController)this.UnityContainer.Resolve(controllerType);
    }
}
```

直接使用本章开始的时候创建的用于管理联系人的 Web API 来演示对自定义 `UnityHttpControllerActivator` 的使用。为了演示 IoC 针对类型的自动映射，我们定义了一个 `IRepository` 接口抽象了对数据的存储。简单起见，仅仅定义了两个获取联系人信息的方法，`DefaultContactRepository` 实现了 `IContactRepository` 接口。

```
public interface IContactRepository
{
    IEnumerable<Contact> GetAllContacts();
    Contact GetContact(string id);
}

public class DefaultContactRepository : IContactRepository
{
    private static List<Contact> contacts = new List<Contact>
    {
        new Contact
        {
            Id = "001",
            Name = "张三",
            PhoneNo = "123",
            EmailAddress = "zhangsan@gmail.com"
        },
        new Contact
        {
            Id = "002",
            Name = "李四",
            PhoneNo = "456",
            EmailAddress = "lisi@gmail.com"
        }
    };

    public IEnumerable<Contact> GetAllContacts()
```

```

    {
        return contacts;
    }

    public Contact GetContact(string id)
    {
        return contacts.FirstOrDefault(c => c.Id == id);
    }
}

```

如下所示的是修改后的 `ContactsController` 的定义，它具有一个应用了 `Dependency Attribute` 特性的依赖属性 `Repository`，其类型为 `IContactRepository`。如果我们利用某个 `UnityContainer` 来创建 `ContactsController` 对象，并且注册了针对 `IContactRepository` 的类型映射以及其他必要的依赖，那么该属性自动被初始化，分别用于获取所有联系人列表和指定联系人的两个 `Action` 方法 `Get` 直接通过这个属性获取数据。

```

public class ContactsController : ApiController
{
    [Dependency]
    public IContactRepository Repository { get; set; }

    public IEnumerable<Contact> Get()
    {
        return this.Repository.GetAllContacts();
    }

    public Contact Get(string id)
    {
        return this.Repository.GetContact(id);
    }
}

```

在 `Global.asax` 中，我们对自定义的 `UnityHttpControllerActivator` 进行了注册。如下面的代码片段所示，我们在 `Application_Start` 方法中创建了一个 `UnityContainer`，并注册了 `IContactRepository` 和 `DefaultContactRepository` 之间的类型映射，最后据此创建一个 `UnityHttpControllerActivator` 对象，并通过 `GlobalConfiguration` 替换掉已经注册的 `HttpControllerActivator`。

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        IUnityContainer unityContainer = new UnityContainer();
        unityContainer.RegisterType<IContactRepository,
            DefaultContactRepository>();
        GlobalConfiguration.Configuration.Services.Replace(
            typeof(IHttpControllerActivator),
            new UnityHttpControllerActivator(unityContainer));
    }
}

```

作为 Web API 宿主的这个 Web 应用运行之后，我们可以直接在浏览器中输入相应的地址获取所有联系人列表和针对某个 ID（001）的联系人信息，具体的输出结果如图 9-11 所示。

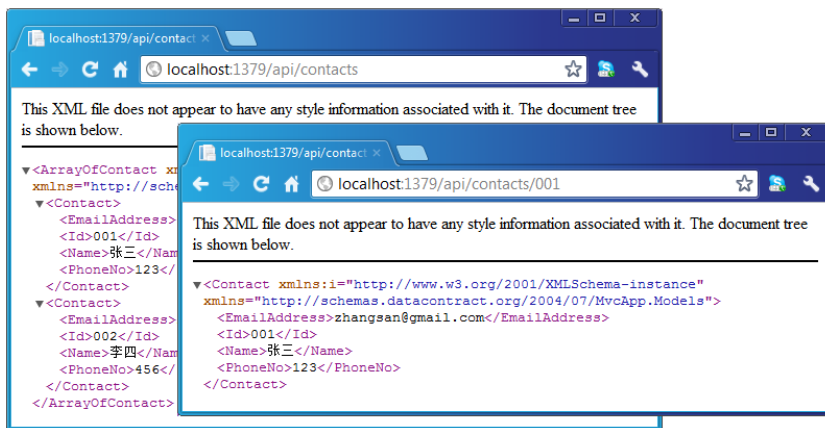


图 9-11 通过浏览器调用 Web API (自定义 ApiControllerActivator)

实例演示：通过自定义 DependencyResolver 实现基于 IoC 的 ApiController 的激活 (S907)

基于 IoC 的 ApiController 激活方式也可以通过自定义的 DependencyResolver 来实现，接下来我们来演示如何通过自定义 DependencyResolver 实现与另一个 IoC 框架 Ninject 的集成，为此我们创建了如下一个类型为 NinjectDependencyResolver 的自定义 DependencyResolver。

```
public class NinjectDependencyResolver : IDependencyResolver
{
    private List<IDisposable> disposableServices = new List<IDisposable>();
    public IKernel Kernel { get; private set; }

    public NinjectDependencyResolver(NinjectDependencyResolver parent)
    {
        this.Kernel = parent.Kernel;
    }

    public NinjectDependencyResolver()
    {
        this.Kernel = new StandardKernel();
    }

    public void Register<TFrom, TTo>() where TTo : TFrom
    {
        this.Kernel.Bind<TFrom>().To<TTo>();
    }

    public IDependencyScope BeginScope()
    {
        return new NinjectDependencyResolver(this);
    }

    public object GetService(Type serviceType)
    {
        var service = this.Kernel.TryGet(serviceType);
    }
}
```

```

        this.AddDisposableService(service);
        return service;
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        foreach (var service in this.Kernel.GetAll(serviceType))
        {
            this.AddDisposableService(service);
            yield return service;
        }
    }

    public void Dispose()
    {
        foreach (IDisposable disposable in disposableServices)
        {
            disposable.Dispose();
        }
    }

    private void AddDisposableService(object servie)
    {
        IDisposable disposable = servie as IDisposable;
        if (null != disposable && !disposableServices.Contains(disposable))
        {
            disposableServices.Add(disposable);
        }
    }
}

```

NinjectDependencyResolver 的核心是类型为 `IKernel` 只读属性 `Kernel`，为了确保获取的服务实例能够被正常地释放，我们定义了一个元素类型为 `IDisposable` 的列表。如果获取的对象实现了 `IDisposable` 接口，它们都会被放入这个列表中，它们会在实现的 `Dispose` 方法中被释放。`BeginScope` 方法返回一个新的 `NinjectDependencyResolver` 对象，它拥有同一个 `Kernel` 对象，针对类型的注册实现在泛型的 `Register<TFrom,TTo>` 方法中。

我们直接将 `NinjectDependencyResolver` 应用于上面这个实例中，只需要将 `Global.asax` 中针对自定义 `HttpControllerActivator` 的注册替换成如下所示的针对 `NinjectDependencyResolver` 的注册即可。

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        NinjectDependencyResolver dependencyResolver =
            new NinjectDependencyResolver();
        dependencyResolver.Register<IContactRepository,
            DefaultContactRepository>();
        GlobalConfiguration.Configuration.DependencyResolver =
            dependencyResolver;
    }
}

```

在上面例子中我们采用的是属性注入方式解决 `ContactsController` 针对 `ContactRepository`

的依赖，现在通过如下的代码切换为构造器注入的方式。我们通过浏览器获取联系人信息依然会得到如图 9-11 所示的结果。

```
public class ContactsController : ApiController
{
    public IContactRepository Repository { get; private set; }
    public ContactsController(IContactRepository repository)
    {
        this.Repository = repository;
    }
    //其他成员
}
```

9.3.2 ApiController 的执行

当 ApiControllerDispatcher 利用 ApiControllerActivator 成功完成针对目标 ApiController 的激活之后，余下的工作基本上体现在对该 ApiController 的执行上。ApiController 实现了具有如下定义的 IHttpController 接口，ExecuteAsync 实现对 ApiController 的执行，该方法返回一个用于处理响应的 Task<HttpResponseMessage> 对象。

```
public interface IHttpController
{
    Task<HttpResponseMessage> ExecuteAsync(
        ApiControllerContext controllerContext,
        CancellationToken cancellationToken);
}
```

ExecuteAsync 方法接受一个 System.Web.Http.Controllers.ApiControllerContext 类型的对象作为其参数，它表示当前的 ApiController 上下文，相当于 ASP.NET MVC 下的 ControllerContext。如下所示的是 ApiControllerContext 的定义，可以通过它得到当前的 ApiController 和描述它的 ControllerDescriptor、代表当前请求的 HttpRequestMessage 对象、当前 HttpConfiguration 和路由数据，以及用于辅助生成 URL 的 UrlHelper 对象。

```
public class ApiControllerContext
{
    public ApiControllerContext();
    public ApiControllerContext(HttpConfiguration configuration,
        IHttpRouteData routeData, HttpRequestMessage request);

    public IHttpController Controller { get; set; }
    public ApiControllerDescriptor ControllerDescriptor { get; set; }
    public HttpRequestMessage Request { get; set; }
    public HttpConfiguration Configuration { get; set; }
    public IHttpRouteData RouteData { get; set; }
    public UrlHelper Url { get; set; }
}
```

表示路由数据的 RouteData 属性类型为 System.Web.Http.Routing.IHttpRouteData，并不是 RouteData。默认情况下该属性返回一个类型为 HostedHttpRouteData 的对象，这是一个内部类型，我们可以简单将它视为对一个 RouteData 对象的封装。Url 属性的类型也不是 ASP.NET

MVC 下的 `UrlHelper`，而是具有如下定义的 `System.Web.Http.Routing.UrlHelper`，方法 `Link` 和 `Route` 根据指定的路由对象注册名称和路由参数解析生成一个 URL，`Link` 方法返回一个绝对地址，`Route` 方法返回相对地址。

```
public class UrlHelper
{
    //其他成员
    public string Link(string routeName,
        IDictionary<string, object> routeValues);
    public string Link(string routeName, object routeValues);

    public string Route(string routeName,
        IDictionary<string, object> routeValues);
    public string Route(string routeName, object routeValues);
}
```

9.3.3 Action 的选择

`HttpController` 的执行最终体现在对某个 Action 方法的执行上，一个 `HttpController` 定了若干 Action 方法，ASP.NET Web API 需要提供一种机制从中选择出一个针对当前请求的 Action。对于 ASP.NET MVC 来说，目标 Action 的名称在经过 URL 路由之后已经保存在了表示路由数据的 `RouteData` 中。ASP.NET Web API 的 Action 选择机制要复杂一些，不过在正式讨论这个问题之前我们先来介绍一下描述 Action 的 `System.Web.Http.Controllers.HttpActionDescriptor` 类型。

HttpActionDescriptor

ASP.NET Web API 中的 Action 通过一个 `HttpActionDescriptor` 对象来描述，它相当于 ASP.NET MVC 中的 `ActionDescriptor`。可以通过 `HttpActionDescriptor` 具有如下定义的属性获取到 Action 相关的描述信息，这包括 Action 的名称、Action 方法的返回类型和用于描述所在 `HttpController` 的 `HttpControllerDescriptor` 对象。`Configuration` 属性表示当前使用的 `HttpConfiguration` 对象，而字典类型的 `Properties` 属性表示附加在 Action 上的一些自定义属性集合。

```
public abstract class HttpActionDescriptor
{
    //其他成员
    public abstract string ActionName { get; }
    public abstract Type ReturnType { get; }
    public HttpControllerDescriptor ControllerDescriptor { get; set; }

    public HttpConfiguration Configuration { get; set; }
    public ConcurrentDictionary<object, object> Properties { get; }

    public virtual HttpActionBinding ActionBinding { get; set; }
    public virtual IActionResultConverter ResultConverter { get; }
    public virtual Collection<HttpMethod> SupportedHttpMethods { get; }
}
```

`ActionBinding` 属性返回的 `HttpActionBinding` 对象实现了参数绑定，我们会在“Action 参数绑定”部分对其进行单独介绍。属性 `ResultConverter` 返回一个 `ActionResultConverter` 对象，它负责将执行 Action 方法返回的对象转化成表示响应的 `HttpResponseMessage` 对象。`SupportedHttpMethods` 属性返回当前 Action 支持的 HTTP 方法列表。

除了上述这些用于描述 Action 的属性外，`HttpActionDescriptor` 还定义了如下一些方法。`ExecuteAsync` 最终实现了对 Action 的执行，而 `GetParameters` 返回一个用于描述 Action 方法参数列表的 `HttpParameterDescriptor` 集合。`GetCustomAttributes` 用于获取应用在 Action 方法上指定类型的特性，而 `GetFilterPipeline` 和 `GetFilters` 用于获取与筛选器相关的信息。

```
public abstract class HttpActionDescriptor
{
    // 其他成员
    public abstract Task<object> ExecuteAsync(
        HttpContext controllerContext,
        IDictionary<string, object> arguments);
    public abstract Collection<HttpParameterDescriptor> GetParameters();

    public virtual Collection<T> GetCustomAttributes<T>() where T: class;
    public virtual Collection<FilterInfo> GetFilterPipeline();
    public virtual Collection<IFilter> GetFilters();
}
```

`HttpActionDescriptor` 是一个抽象类，ASP.NET Web API 默认使用的具体类型是继承它的 `System.Web.Http.Controllers.ReflectedHttpActionDescriptor`，它通过对 Action 方法实施反射得到相关的描述信息。

Action 对 HTTP 方法的支持

由于 REST 架构往往将 HTTP 方法视为针对资源的操作类型，所以一个 Action 一般都对应着某一种 HTTP 方法，只有具备匹配 HTTP 方法的请求才能访问它，不过我们可以通过在 Action 方法上应用相应的特性使之可以支持多种 HTTP 方法。

在默认的情况下，ASP.NET Web API 利用 Action 方法的名称来决定其支持的 HTTP 方法，如果 Action 方法名称将某种 HTTP 方法名称（GET、POST、PUT、DELETE、HEAD、OPTIONS 和 PATCH）作为前缀（不区分大小写），那么对应的 HTTP 方法就是 Action 唯一支持的 HTTP 方法，否则默认支持的 HTTP 方法是 POST。

我们可以通过一个很简单的实例演示来证实这一点，在一个 ASP.NET MVC 应用中定义如下一个 `DemoController`，对于定义其中的 8 个 Action 方法，除了最后一个 `Other` 外其余都采用相应的 HTTP 方法名作为前缀。

```
public class DemoController : ApiController
{
    public void GetXxx() { }
    public void PostXxx() { }
    public void PutXxx() { }
    public void DeleteXxx() { }
}
```

```

public void HeadXxx() { }
public void OptionsXxx() { }
public void PatchXxx() { }
public void Other() { }
}

```

然后我们定义了如下一个 `HomeController`。辅助方法 `GetActionDescriptors` 用于根据指定的 `ApiController` 类型得到用于描述所有 Action 的 `HttpActionDescriptor` 列表, 在默认的方法 `Index` 中, 我们调用该方法获取用于描述定义在 `DemoController` 中所有 Action 的 `HttpActionDescriptor` 列表, 并将其作为 Model 呈现在默认的 View 中。

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View(this.GetActionDescriptors(typeof(DemoController)));
    }

    private IEnumerable<HttpActionDescriptor> GetActionDescriptors(
        Type controllerType)
    {
        ApiControllerDescriptor controllerDescriptor =
            new ApiControllerDescriptor();
        controllerDescriptor.ControllerType = controllerType;
        MethodInfo[] methods = controllerType.GetMethods(
            BindingFlags.InvokeMethod | BindingFlags.Public |
            BindingFlags.Instance);
        foreach (var method in methods)
        {
            if (method.GetBaseDefinition().DeclaringType
                .IsAssignableFrom(typeof(ApiController)))
            {
                continue;
            }
            yield return new ReflectedHttpActionDescriptor(
                controllerDescriptor, method);
        }
    }
}

```

如下所示的是 Action 方法 `Index` 对应 View 的定义, 这是一个 Model 类型为 `IEnumerable<HttpActionDescriptor>` 的强类型 View。在该 View 中我们通过调用每个 `HttpActionDescriptor` 的 `SupportedHttpMethods` 属性得到对应 Action 所支持的 HTTP 方法, 并以表格的形式呈现出来。

```

@model IEnumerable<HttpActionDescriptor>
<html>
    <head>
        <title>Action 对 HTTP 方法的支持</title>
    </head>
    <body>
        <table rules="all">
            <tr><th>Action Name</th><th>HTTP Method</th></tr>
            @foreach (var action in Model)
            {
                var httpMethods = action.SupportedHttpMethods.ToArray();

```

```

<tr>
  <td rowspan="@httpMethods.Length">@action.ActionName</td>
  <td>@httpMethods[0]</td>
</tr>
for (int i = 1; i < httpMethods.Length; i++)
{
  <tr><td>@httpMethods[i]</td></tr>
}
}
</table>
</body>
</html>

```

上面这个程序运行之后会在浏览器中呈现出如图 9-12 所示的输出结果，从中可以看出对应定义在 DemoController 的 8 个 Action，除了 Other 采用默认支持 HTTP-POST 方法外，其余 7 个支持的 HTTP 方法的都由方法名称前缀决定。（S908）

Action Name	HTTP Method
GetXxx	GET
PostXxx	POST
PutXxx	PUT
DeleteXxx	DELETE
HeadXxx	HEAD
OptionsXxx	OPTIONS
PatchXxx	PATCH
Other	POST

图 9-12 基于 Action 的 HTTP 方法支持规则

如果我们不希望某个 Action 支持由它名称决定的 HTTP 方法，或者希望某个 Action 支持多种不同的 HTTP 方法，我们可以在 Action 方法上面应用针对某种 HTTP 方法的特性。上述的 7 种 HTTP 方法均具有相应的特性 (HttpGetAttribute、HttpPostAttribute、HttpPutAttribute、HttpDeleteAttribute、HttpHeadAttribute、HttpOptionsAttribute、HttpPatchAttribute)。这些特性类型均定义在命名空间 System.Web.Http 下，请读者注意与定义在 System.Web.Mvc 命名空间下的同名特性相区分。除了这些基于某种 HTTP 方法的特性，我们还可以利用具有如下定义的 AcceptVerbsAttribute 以字符串数组的形式指定支持的 HTTP 方法列表。

```

[AttributeUsage(AttributeTargets.Method, AllowMultiple=false, Inherited=true)]
public sealed class AcceptVerbsAttribute : Attribute, IActionHttpMethodProvider
{
    public AcceptVerbsAttribute(params string[] methods);
    public Collection<HttpMethod> HttpMethods { get; }
}

```

为了演示应用这些特性导致对 Action 支持的 HTTP 方法的改变，我们将上面定义的 DemoController 的每个 Action 方法都应用了 HttpGetAttribute 和 HttpPostAttribute 特性。

```

public class DemoController : ApiController
{

```

```

[HttpGet]
[HttpPost]
public void GetXxx() { }

[HttpGet]
[HttpPost]
public void PostXxx() { }

[HttpGet]
[HttpPost]
public void PutXxx() { }

[HttpGet]
[HttpPost]
public void DeleteXxx() { }

[HttpGet]
[HttpPost]
public void HeadXxx() { }

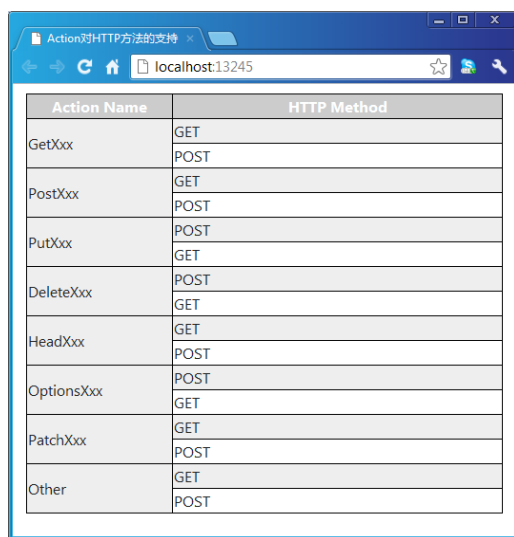
[HttpGet]
[HttpPost]
public void OptionsXxx() { }

[HttpGet]
[HttpPost]
public void PatchXxx() { }

[HttpGet]
[HttpPost]
public void Other() { }
}

```

再次运行程序，将会得到如图 9-13 所示的输出结果，可以看出所有的 Action 均只支持 GET 和 POST 这两种 HTTP 方法。（S909）



Action Name	HTTP Method
GetXxx	GET
	POST
PostXxx	GET
	POST
PutXxx	POST
	GET
DeleteXxx	POST
	GET
HeadXxx	GET
	POST
OptionsXxx	POST
	GET
PatchXxx	GET
	POST
Other	GET
	POST

图 9-13 通过特性应用决定 Action 支持的 HTTP 方法

HttpActionSelector

ASP.NET Web API 采用 `HttpActionSelector` 这个组件根据当前请求实现对目标 `Action` 的选择。`HttpActionSelector` 实现了具有如下定义的接口 `System.Web.Http.Controllers.IHttpActionSelector`，方法 `GetActionMapping` 返回一个 `ILookup<string, HttpActionDescriptor>` 对象，它体现了 `Action` 名称与描述对应 `Action` 的 `HttpActionDescriptor` 对象（一个或者多个）之间的匹配关系，真正对 `Action` 的选择实现在 `SelectAction` 方法中。

```
public interface IHttpActionSelector
{
    ILookup<string, HttpActionDescriptor> GetActionMapping(
        HttpControllerDescriptor controllerDescriptor);
    HttpActionDescriptor SelectAction(HttpControllerContext controllerContext);
}
```

因为一个 `Action` 名可以同时匹配多个 `Action` 方法，所以 `GetActionMapping` 方法返回的是一个 `ILookup<string, HttpActionDescriptor>` 对象。以本章开始演示的“联系人管理”Web API 为例，对于定义在 `ContactsController` 的 5 个 `Action`，它们与 `Action` 名称的映射关系如表 9-2 所示。

表 9-2 Action 名称与 Action 方法的映射关系

Action 名称	Action 方法
Get	IEnumerable<Contact>Get()Get(string id)
Put	void Put(Contactcontact)
Post	void Post(Contact contact)
Delete	void Delete(string id)

通过 `GetActionMapping` 方法的 `HttpActionDescriptor` 列表只包括定义在 `HttpController` 的公有的实例方法，并且不包括从基类 `ApiController` 继承下来的方法。如果我们在 `Action` 方法上应用 `System.Web.Http.NonActionAttribute` 特性，对应的 `HttpActionDescriptor` 依然会出现在列表中。

ASP.NET Web API 依然是利用 `ServicesContainer` 以接口的方式来获取注册的 `HttpActionSelector` 服务对象，我们可以直接通过 `ServicesContainer` 的扩展方法 `GetActionSelector` 来获取该对象。默认情况下注册的 `HttpActionSelector` 类型为 `System.Web.Http.Controllers.ApiControllerActionSelector` 对象。下面的代码片段给出了扩展方法 `GetActionSelector` 和 `ApiControllerActionSelector` 类型的定义。

```
public static class ServicesExtensions
{
    //其他成员
    public static IHttpActionSelector GetActionSelector(
        this ServicesContainer services);
}
```

```
public class ApiControllerActionSelector : IHttpActionSelector
{
    public virtual ILookup<string, HttpActionDescriptor>
        GetActionMapping(HttpControllerDescriptor controllerDescriptor);
    public virtual HttpActionDescriptor SelectAction(
        HttpContext controllerContext);
}
```

实现在 `ApiControllerActionSelector` 的 `SelectAction` 方法采用如下的步骤根据提供的 `HttpContext` 选择出正确的 `Action`。

- 步骤一：如果当前 `HttpContext` 中的路由数据包含目标 `Action` 的名称，则调用它的 `GetActionMapping` 方法得到 `Action` 名称和 `HttpActionDescriptor` 的映射集合，然后根据该名称从中筛选出匹配的 `HttpActionDescriptor` 列表，并剔除不支持当前请求 HTTP 方法的 `HttpActionDescriptor`。如果 `Action` 名称在路由数据中找不到，它会从所有的 `HttpActionDescriptor` 中筛选出支持当前请求 HTTP 方法的 `HttpActionDescriptor` 列表。我们将筛选的 `HttpActionDescriptor` 列表称为 `List1`。
- 步骤二：遍历 `List1` 中的所有 `HttpActionDescriptor`，通过调用 `GetParameters` 方法得到用于描述对应参数列表的一组 `HttpParameterDescriptor` 对象，并进一步提取参数名称列表。然后提取路由数据中所有路由变量名和当前请求查询字符串的 `Key`，并根据它们从 `List1` 中选择出参数名称列表与之相匹配的 `HttpActionDescriptor`。我们将这个列表称为 `List2`。
- 步骤三：如果对应的 `Action` 方法上应用了 `NonActionAttribute` 特性，我们将相应的 `HttpActionDescriptor` 从 `List2` 中剔除，得到最终列表 `List3`。如果 `List3` 包含一个唯一的 `HttpActionDescriptor`，它将会作为 `SelectAction` 方法的返回值。反之，如果该列表为空或者包含多个 `HttpParameterDescriptor`，则分别抛出 HTTP 状态为“404, Not Found”和“500, Internal Error”的 `HttpResponseException` 异常。

按照上面介绍的 `Action` 选择机制，如果我们调用 `ContactsController` 的 `Get` 方法获取指定 ID 的联系人信息，请求的 URL 可以是“`/Api/Contacts/001`”，也可以是“`/Api/Contacts?id=001`”。

9.3.4 Model 元数据的解析

目标 `Action` 被选择出来后，照理说接下来的步骤就是执行 `Action` 了，但是执行 `Action` 方法需要初始化输入参数列表，而参数绑定又需要使用到 `Model` 元数据信息，所以我们先来介绍一下 `Model` 元数据的解析问题。由于 ASP.NET Web API 采用了与 ASP.NET MVC 几乎一样的 `Model` 元数据解析机制，所以我们在这里只做一些概括性的介绍。

ASP.NET Web API 下的 `Model` 元数据通过类型 `System.Web.Http.Metadata.ModelMetadata` 表示。`ModelMetadata` 具有一个树型层级结构，不仅仅用于描述数据类型本身，还用于描述其属性成员。`ModelMetadata` 的层级结构直接体现在如下所示的类型定义上，它具有一个

类型为 `IEnumerable<ModelMetadata>` 的只读属性 `Properties`，用于表示所有属性的 `Model` 元数据。

```
public class ModelMetadata
{
    //其他成员
    public virtual IEnumerable<ModelMetadata> Properties { get; }
}
```

与 ASP.NET MVC 下的 `ModelMetadata` 相比较，ASP.NET Web API 中的 `ModelMetadata` 的属性成员要少得多。`ModelMetadata` 具有四个与类型相关的只读属性：`ModelType` 和 `ContainerType` 属性表示 `Model` 自身的属性和容器类型；`IsComplexType` 属性表示 `Model` 是否是一个复杂类型，简单类型和复杂类型的区别在类型本身是否支持源自字符串类型的转换；`IsNullableValueType` 属性则表示是否属于可空值类型。

```
public class ModelMetadata
{
    //其他成员
    public Type           ModelType { get; }
    public Type           ContainerType { get; }
    public virtual bool    IsComplexType { get; }
    public bool           IsNullableValueType { get; }

    public string         PropertyName { get; }
    public virtual string Description { get; set; }
    public virtual bool    IsReadOnly { get; set; }
    public object         Model { get; set; }
    public virtual bool    ConvertEmptyStringToNull { get; set; }

    public virtual Dictionary<string, object> AdditionalValues { get; }
}
```

如果 `ModelMetadata` 是针对容器类型的某个属性，其属性 `PropertyName` 返回对应属性的名称。`Description` 属性用于返回和设置相应的描述性文字，而属性 `IsReadOnly` 用于控制其读写性。真正的数据值通过 `Model` 属性来获取或者设置。布尔类型的 `ConvertEmptyStringToNull` 表示是否需要将空字符串视为 `Null`，而字典类型的 `AdditionalValues` 属性用于自定义一些额外的属性。

`ModelMetadata` 还定义了一个类型为 `ModelMetadataProvider` 的 `Provider` 属性作为 `Model` 元数据的提供者，该属性也可以在构造函数中进行初始化，方法 `GetDisplayName` 用于返回显示名称，而 `GetValidators` 则返回用于进行 `Model` 验证的 `ModelValidator` 列表。

```
public class ModelMetadata
{
    //其他成员
    protected ModelMetadataProvider Provider { get; set; }
    public ModelMetadata(ModelMetadataProvider provider, Type containerType,
        Func<object> modelAccessor, Type modelType, string propertyName);

    public string GetDisplayName();
    public virtual IEnumerable<ModelValidator>
        GetValidators(IEnumerable<ModelValidatorProvider> validatorProviders);
}
```


与 ASP.NET MVC 一样, ASP.NET Web API 同样采用基于数据注解特性的元数据定制方式, 具体使用的特性包括 `DisplayAttribute`、`DisplayFormatAttribute`、`EditableAttribute` 和 `ReadOnlyAttribute`。这四个特性作为属性定义在具有如下定义的类型 `CachedDataAnnotationsMetadataAttributes` 中, 该类型定义在 `System.Web.Http.Metadata.Providers` 的命名空间下。

```
public class CachedDataAnnotationsMetadataAttributes
{
    public CachedDataAnnotationsMetadataAttributes(Attribute[] attributes);

    public DisplayAttribute      Display { get; }
    public DisplayFormatAttribute DisplayFormat { get; }
    public EditableAttribute      Editable { get; }
    public ReadOnlyAttribute      ReadOnly { get; }
}
```

`System.Web.Http.Metadata.Providers.CachedDataAnnotationsModelMetadata` 是 ASP.NET Web API 用于描述 Model 元数据的具体类型, 而它的基类则是另一个继承自 `ModelMetadata` 的抽象类型 `System.Web.Http.Metadata.Providers.CachedModelMetadata<TPrototypeCache>` (它具体继承的基类是 `CachedModelMetadata<CachedDataAnnotationsMetadataAttributes>`)。通过第4章“Model 元数据的解析”的介绍我们知道, 在 ASP.NET MVC 的 Model 元数据系统中均定义着一套同名的类型。

至于用于提供 Model 元数据的 `ModelMetadataProvider`, 它们最终都继承自具有如下定义的抽象类 `System.Web.Http.Metadata.ModelMetadataProvider`, 它与 ASP.NET MVC 中的同名类型具有一致的定义。如下面的代码片段所示, 它定义了三个用于提供 Model 元数据的抽象方法, 其中 `GetMetadataForType` 根据指定的数据类型得到相应的 Model 元数据, 而 `GetMetadataForProperties` 和 `GetMetadataForProperty` 则用于获取容器类型的属性相关的 Model 元数据, 前者返回针对所有属性的 `ModelMetadata` 集合, 后者返回指定属性的 `ModelMetadata`。

```
public abstract class ModelMetadataProvider
{
    public abstract ModelMetadata GetMetadataForType(Func<object> modelAccessor,
        Type modelType);
    public abstract IEnumerable<ModelMetadata> GetMetadataForProperties(
        object container, Type containerType);
    public abstract ModelMetadata GetMetadataForProperty(
        Func<object> modelAccessor, Type containerType, string propertyName);
}
```

`CachedDataAnnotationsModelMetadata` 最终通过 `CachedDataAnnotationsModelMetadataProvider` 来提供, 后者又继承自抽象类型 `CachedAssociatedMetadataProvider<TModelMetadata>`。而 `AssociatedMetadataProvider` 又是 `CachedAssociatedMetadataProvider<TModelMetadata>` 的基类。这些类型均定义在 `System.Web.Http.Metadata.Providers` 命名空间下, 我们在 ASP.NET MVC 的 Model 元数据系统中可以看到完全一致的类型定义。

ASP.NET Web API 使用的 `ModelMetadataProvider` 同样也是注册在当前 `HttpConfiguration` 的 `ServicesContainer` 中，我们可以调用 `ServicesContainer` 具有如下定义的扩展方法 `GetModelMetadataProvider` 来得到注册的 `ModelMetadataProvider` 对象。

```
public static class ServicesExtensions
{
    // 其他成员
    public static ModelMetadataProvider GetModelMetadataProvider(
        this ServicesContainer services);
}
```

9.3.5 Action 参数绑定

在执行目标 `Action` 方法之前需要从请求中提取相应的数据作为参数的值，ASP.NET MVC 通过 `Model` 绑定的方式实现了对参数的绑定，但是对于 ASP.NET Web API 来说，`Model` 绑定仅仅是参数绑定的一种方式而已。

在调用 `Action` 方法之前，表示 `Action` 上下文的 `System.Web.Http.Controllers.HttpActionContext` 对象被创建出来。如下面的代码片段所示，`HttpActionContext` 具有一个表示作为 `Action` 方法参数列表的 `ActionArguments` 属性。该属性返回的是一个字典对象，`Key` 和 `Value` 分别表示参数名和参数值，参数的绑定目标就是填充这个字典。

```
public class HttpActionContext
{
    public HttpActionContext();
    public HttpActionContext(HttpControllerContext controllerContext,
        HttpActionDescriptor actionDescriptor);

    public Dictionary<string, object> ActionArguments { get; }
    public HttpActionDescriptor ActionDescriptor { get; set; }
    public HttpControllerContext ControllerContext { get; set; }
    public ModelStateDictionary ModelState { get; }
    public HttpRequestMessage Request { get; }
    public HttpResponseMessage Response { get; set; }
}
```

除 `ActionArguments` 之外，通过 `HttpActionContext` 还可以得到其他上下文信息，包括用于描述 `Action` 的 `HttpActionDescriptor` 对象、表示当前 `HttpController` 上下文的 `HttpControllerContext` 对象、表示当前请求/响应的 `HttpRequestMessage/HttpResponseMessage` 等。

HttpParameterDescriptor

针对 `Action` 方法某个参数的绑定依赖于描述参数的元数据信息。ASP.NET Web API 定义了一个具有如下定义的 `System.Web.Http.Controllers.HttpParameterDescriptor` 类型来描述 `Action` 方法的参数，`HttpActionDescriptor` 的 `GetParameters` 方法获取的就是一个 `HttpParameterDescriptor` 对象列表。

```

public abstract class HttpParameterDescriptor
{
    protected HttpParameterDescriptor();
    protected HttpParameterDescriptor(HttpActionDescriptor actionDescriptor);
    public virtual Collection<T> GetCustomAttributes<T>() where T : class;

    public abstract string      ParameterName { get; }
    public abstract Type        ParameterType { get; }
    public virtual object       DefaultValue { get; }

    public virtual ModelBinderAttribute ModelBinderAttribute { get; set; }
    public virtual string Prefix { get; }
    public HttpActionDescriptor ActionDescriptor { get; set; }
    public HttpConfiguration Configuration { get; set; }

    public ConcurrentDictionary<object, object> Properties { get; }
}

```

如上面的代码片段所示，通过 `HttpParameterDescriptor` 不仅可以得到参数的名称、类型和默认值，还可以获取用于指定 `ModelBinder` 类型的 `ModelBinderAttribute` 特性和绑定前缀。属性 `ActionDescriptor` 返回描述所在 `Action` 的 `HttpActionDescriptor` 对象，而 `Configuration` 属性自然返回当前的 `HttpConfiguration` 对象。通过属性 `Properties` 我们可以将作为参数属性的任意类型的对象附加到 `HttpParameterDescriptor` 上，除此之外，通过调用 `GetCustomAttributes` 方法还可以获取应用到参数上的指定类型的特性列表。

`HttpParameterDescriptor` 是一个抽象类，ASP.NET Web API 具体使用的是一个具有如下定义的 `System.Web.Http.Controllers.ReflectedHttpParameterDescriptor` 类型，它具有一个 `ParameterInfo` 类型的属性，重写的三个属性（`DefaultValue`、`ParameterName` 和 `ParameterType`）直接来源于该对象。

```

public class ReflectedHttpParameterDescriptor : HttpParameterDescriptor
{
    public ReflectedHttpParameterDescriptor();
    public ReflectedHttpParameterDescriptor(
        HttpActionDescriptor actionDescriptor, ParameterInfo parameterInfo);
    public override Collection<T> GetCustomAttributes<T>() where T : class;

    public override object      DefaultValue { get; }
    public override string      ParameterName { get; }
    public override Type        ParameterType { get; }
    public ParameterInfo         ParameterInfo { get; set; }
}

```

HttpActionBinding-HttpParameterBinding-ActionValueBinder

ASP.NET Web API 实现对目标 `Action` 参数绑定的是一个名为 `HttpActionBinding` 的组件，对应的类型为具有如下定义的 `System.Web.Http.Controllers.HttpActionBinding`。`HttpActionBinding` 本身并不真正地实施对具体参数的绑定，它将这个工作交给另一个名为 `HttpParameterBinding` 的组件来完成。

```

public class HttpActionBinding
{
    public HttpActionBinding();
    public HttpActionBinding(HttpActionDescriptor actionDescriptor,
        HttpParameterBinding[] bindings);
    public virtual Task ExecuteBindingAsync(HttpActionContext actionContext,
        CancellationToken cancellationToken);

    public HttpActionDescriptor      ActionDescriptor { get; set; }
    public HttpParameterBinding[]    ParameterBindings { get; set; }
}

```

如上面的代码片段所示，`HttpActionBinding` 具有一个类型为 `HttpParameterBinding` 数组的属性 `ParameterBindings`，目标 `Action` 方法的每一个参数对应着一个 `HttpParameterBinding` 对象，而针对该参数的绑定工作就由它来完成。`HttpActionBinding` 的虚方法 `ExecuteBindingAsync` 返回一个用于执行参数列表绑定的 `Task` 对象。

对单个参数实施绑定的 `HttpParameterBinding` 继承自具有如下定义的抽象类型 `System.Web.Http.Controllers.HttpParameterBinding`。抽象方法 `ExecuteBindingAsync` 返回一个完成参数绑定的 `Task` 对象，由于在进行绑定过程中往往需要使用到描述参数类型的 `ModelMetadata`，所以该方法接受一个用于提供 `ModelMetadata` 的 `ModelMetadataProvider` 作为参数。

```

public abstract class HttpParameterBinding
{
    protected HttpParameterBinding(HttpParameterDescriptor descriptor);
    public abstract Task ExecuteBindingAsync(
        ModelMetadataProvider metadataProvider, HttpActionContext actionContext,
        CancellationToken cancellationToken);

    public HttpParameterDescriptor Descriptor { get; }
    public virtual string ErrorMessage { get; }
    public bool IsValid { get; }

    public virtual bool WillReadBody { get; }
}

```

`HttpParameterBinding` 的只读属性 `Descriptor` 返回用于描述参数的 `HttpParameterDescriptor` 对象，该对象在构造函数中进行初始化。与 ASP.NET MVC 类似，ASP.NET Web API 在参数绑定过程中会对得到的参数值实施验证，`IsValid` 属性用于表示验证是否成功通过，而具体的错误消息通过 `ErrorMessage` 属性返回。另一个只读属性 `WillReadBody` 表示绑定参数值的数据是否需要通过读取请求消息的主体部分来获取，该属性直接返回 `False`。

ASP.NET Web API 定义了如下所示的一系列的 `HttpParameterBinding` 类型用于实现针对不同数据来源的参数绑定。如果它们不能满足具体需求，我们还可以自定义 `HttpParameterBinding`。

- `CancellationTokenParameterBinding`：它直接将调用 `ExecuteBindingAsync` 方法传入的 `CancellationToken` 类型参数 `cancellationTok` 作为相应的参数值。

- **ErrorParameterBinding**: 它直接抛出一个 `InvalidOperationException` 异常。
- **FormatterParameterBinding**: 它会读取请求消息主体的内容, 并利用匹配的消息格式化器对读取内容进行反序列化来生成作为参数值的对象。
- **HttpRequestParameterBinding**: 它直接将表示当前请求的 `HttpRequestMessage` 对象作为相应的参数值。
- **ModelBinderParameterBinding**: 它利用提供的 `ModelBinder` 来创建作为参数值的对象。

`HttpActionBinding` 通过 `HttpParameterBinding` 来完成参数的绑定, 而 `HttpActionBinding` 对象本身则是通过另一个名为 `ActionValueBinder` 的组件来提供的。所有的 `ActionValueBinder` 都实现了具有如下定义的接口 `System.Web.Http.Controllers.IActionValueBinder`, 该接口具有唯一的方法 `GetBinding`, 该方法根据指定的 `HttpActionDescriptor` 对象返回对应 `HttpActionBinding` 的对象。

```
public interface IActionValueBinder
{
    HttpActionBinding GetBinding(HttpActionDescriptor actionDescriptor);
}
```

同其他的组件一样, `ActionValueBinder` 通过可以注册在当前 `HttpConfiguration` 的 `ServicesContainer` 中, 我们可以调用 `ServicesContainer` 具有如下定义的扩展方法 `GetActionValueBinder` 来获取注册的 `ActionValueBinder` 对象。

```
public static class ServicesExtensions
{
    //其他成员
    public static IActionValueBinder GetActionValueBinder(
        this DefaultServices services);
}
```

默认注册的 `ActionValueBinder` 类型为 `System.Web.Http.ModelBinding.DefaultActionValueBinder`。如下面的代码片段所示, `DefaultActionValueBinder` 除了重写抽象方法 `GetBinding` 之外, 它还定义了两个受保护的虚方法 `GetBodyModelValidator` 和 `GetFormatters`, 这两个方法涉及到对参数的验证和消息的格式化, 我们将在后续部分对其进行介绍。

```
public class DefaultActionValueBinder : IActionValueBinder
{
    public virtual HttpActionBinding GetBinding(
        HttpActionDescriptor actionDescriptor);

    protected virtual IBodyModelValidator GetBodyModelValidator(
        HttpActionDescriptor actionDescriptor);
    protected virtual IEnumerable<MediaTypeFormatter> GetFormatters(
        HttpActionDescriptor actionDescriptor);
}
```

ASP.NET Web API 的 Action 参数绑定系统由 `HttpActionBinding`、`HttpParameterBinding` 和 `ActionValueProvider` 这三个核心组件构成, 如图 9-14 所示的 UML 包含了与之相关的接口和类型, 我们可以从这个 UML 中看出这些类型和接口之间的关系。

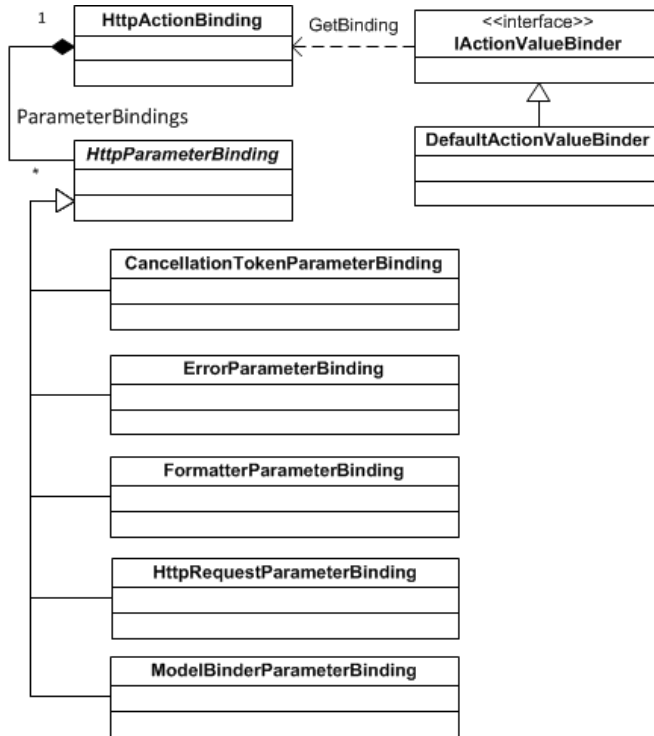


图 9-14 HttpActionBinding、HttpParameterBinding 和 ActionValueProvider

FormatterParameterBinding

具有如下定义的 `FormatterParameterBinding` 是一个重要的 `HttpParameterBinding`，它会读取 HTTP 请求消息主体的内容，并利用匹配的格式化器对读取内容进行反序列化生成相应的参数对象。因为由它绑定的参数值来源于请求消息的主体，所以被重写的属性 `WillReadBody` 返回 `True`。属性 `Formatters` 返回一个用于反序列化读取内容并生成参数对象的格式化器列表，而另一个 `BodyModelValidator` 属性与输入验证有关，我们将在后续部分对它进行单独介绍。

```

public class FormatterParameterBinding : HttpParameterBinding
{
    public FormatterParameterBinding(HttpParameterDescriptor descriptor,
        IEnumerable<MediaTypeFormatter> formatters,
        IBodyModelValidator bodyModelValidator);
    public override Task ExecuteBindingAsync(
        ModelMetadataProvider metadataProvider, HttpContext actionContext,
        CancellationToken cancellationToken);
    protected virtual Task<object> ReadContentAsync(HttpRequestMessage request,
        Type type, IEnumerable<MediaTypeFormatter> formatters,
        IFormatterLogger formatterLogger);

    public IBodyModelValidator BodyModelValidator { get; set; }
    public IEnumerable<MediaTypeFormatter> Formatters { get; set; }
    public override bool WillReadBody { get; }
}
  
```

表示 HTTP 请求/响应的 `HttpRequestMessage/HttpResponseMessage` 具有一个类型为 `HttpContent` 的属性 `Content`, 通过 `HttpContent` 可以读取消息主体的内容并获取与内容相关的 HTTP 报头。当 `FormatterParameterBinding` 在进行参数绑定的时候, 如果当前 `HttpRequestMessage` 的 `Content` 属性是一个 `System.Net.Http.ObjectContent` 对象, 意味着主体的内容已经是一个具体的对象了, 那么它会将被封装的对象直接作为绑定的参数值。否则会调用其 `ReadAsStreamAsync` 方法读取消息主体的内容, 并利用相应的格式化器对其进行反序列化。

这里对读取的主体内容进行反序列化的格式化器被称为 `MediaTypeFormatter`, 通过具有如下定义的 `System.Net.Http.Formatting.MediaTypeFormatter` 类型表示。实际上 `MediaTypeFormatter` 不仅仅用于针对请求消息的反序列化以生成对应的参数对象, `Action` 方法执行的结果被写入响应消息所需的序列化也是通过它来完成的。序列化和反序列化操作分别通过 `WriteToStreamAsync` 和 `ReadFromStreamAsync` 来完成。另外两个抽象方法 `CanWriteType` 和 `CanReadType` 用于判断当前的格式化器是否支持针对指定数据类型的序列化和反序列化。

```
public abstract class MediaTypeFormatter
{
    //其他成员
    public abstract bool CanReadType(Type type);
    public abstract bool CanWriteType(Type type);

    public virtual Task<object> ReadFromStreamAsync(Type type, Stream stream,
        HttpContentHeaders contentHeaders, IFormatterLogger formatterLogger);
    public virtual Task WriteToStreamAsync(Type type, object value,
        Stream stream, HttpContentHeaders contentHeaders,
        TransportContext transportContext);

    public Collection<MediaTypeHeaderValue> SupportedMediaTypes { get; }
}
```

从命名可以看出, `MediaTypeFormatter` 是针对某种媒体类型的格式化器, 或者说它提供的序列化/反序列化功能只能适用于某种具体的媒体类型, 而它的只读属性 `SupportedMediaTypes` 表示支持的媒体类型列表。 `MediaTypeFormatter` 是一个抽象类, 我们常用的 `MediaTypeFormatter` 包括如下四类。

- `FormUrlEncodedMediaTypeFormatter`: 它用于实现对提交的表单进行反序列化, 对应的媒体类型为 “application/x-www-form-urlencoded”。由于表单仅仅限于请求消息, 所以它不支持针对响应消息的序列化。
- `JQueryMvcFormUrlEncodedFormatter`: 它继承自 `FormUrlEncodedMediaTypeFormatter`, 两者不同之处在于, `FormUrlEncodedMediaTypeFormatter` 仅仅用于针对类型为 `FormCollection` 参数的反序列化, 而 `JQueryMvcFormUrlEncodedFormatter` 则没有此限制。
- `JsonMediaTypeFormatter`: 它以 JSON 的格式分别对请求消息和响应消息实施反序列化和序列化, 对应的媒体类型包括 “application/json” 和 “text/json”。
- `XmlMediaTypeFormatter`: 它以 XML 的格式分别对请求消息和响应消息实施反序列化和序列化, 对应的媒体类型包括 “application/xml” 和 “text/xml”。

FormatterParameterBinding 通过它的 Formatters 属性维护着一组 MediaTypeFormatter 列表，它根据请求消息的媒体类型来选择匹配的 MediaTypeFormatter。作为参数数据源的内容从请求消息主体部分读出来之后，会交给这个被选择出来的 MediaTypeFormatter 进行反序列化并最终生成对应的参数对象。被选择的 MediaTypeFormatter 除了支持当前请求的媒体类型之外，还需要支持针对参数类型的反序列化，后者通过调用 CanReadType 方法来检测。

在默认的情况下，ASP.NET Web API 创建的 FormatterParameterBinding 包含的 MediaTypeFormatter 列表来源于当前 HttpConfiguration，具体来说来源于 HttpConfiguration 如下所示的只读属性 Formatters，这是一个元素类型为 MediaTypeFormatter 的集合。当 HttpConfiguration 被初始化的时候，ASP.NET Web API 会创建上述四种类型的 MediaTypeFormatter 并添加到该列表中。我们可以利用 HttpConfiguration 来注册自定义的 MediaTypeFormatter。

```
public class HttpConfiguration : IDisposable
{
    //其他成员
    public MediaTypeFormatterCollection Formatters { get; }
}
```

ModelBinderParameterBinding

对于 ASP.NET MVC 来说，执行目标 Action 方法传入的参数都是通过 Model 绑定的方式进行绑定的，ASP.NET Web API 也通过 ModelBinderParameterBinding 提供了类似的 Model 绑定机制。由于在第 5 章中已经对 ASP.NET MVC 的 Model 绑定作了非常详尽的介绍，在这里我们只对 Model 绑定在 ASP.NET Web API 中的实现机制作一个简单的讲述。ASP.NET Web API 的 Model 绑定涉及的类型大都定义在 System.Web.Http.ModelBinding 命名空间下，请不要与定义在 System.Web.Mvc 命名空间下的同名类型相混淆。

ModelBinder 是 ASP.NET Web API 的 Model 绑定系统中最为核心的组件，它实现了具有如下定义的 IModelBinder 接口，Model 绑定实现在唯一的方法 BindModel 方法中。该方法具有两个上下文对象作为参数，一个是基于当前 Action 的 HttpContext，另一个是针对当前 Model 绑定的 ModelBindingContext。ModelBindingContext 具有一个属性 Model 表示需要绑定的 Model 对象，ModelBinder 的最终目的就在于对 ModelBindingContext 的这个属性赋值。如果 BindModel 方法成功地为当前 ModelBindingContext 绑定了一个 Model 对象，该方法返回 True，否则返回 False。

```
public interface IModelBinder
{
    bool BindModel(HttpContext actionContext,
        ModelBindingContext bindingContext);
}

public class ModelBindingContext
{
    //其他成员
    public object Model { get; set; }
}
```


`ModelBinder` 通过对应的 `ModelBinderProvider` 来提供, 后者继承自具有如下定义的抽象类型 `ModelBinderProvider`, 它的 `GetBinder` 方法会根据当前两个上下文对象得到对应的 `ModelBinder`。ASP.NET MVC 中无此抽象类型, 只有一个 `IModelBinderProvider` 接口。ASP.NET Web API 中定义了一系列具体的 `ModelBinder` 类型, 它们均具有对应的 `ModelBinderProvider`。

```
public abstract class ModelBinderProvider
{
    public abstract IModelBinder GetBinder(HttpContext actionContext,
        ModelBindingContext bindingContext);
}
```

ASP.NET Web API 使用的 `ModelBinderProvider` 依然被注册在当前 `HttpConfiguration` 的 `ServicesContainer` 中, 我们可以通过它具有如下定义的扩展方法 `GetModelValidatorProviders` 得到注册的 `ModelBinderProvider` 列表。下面的代码片段给出了 `DefaultServices` 构造函数的部分定义, 可以看出 ASP.NET Web API 为我们预先注册了一系列默认的 `ModelBinderProvider` 对象。

```
public static class ServicesExtensions
{
    //其他成员
    public static IEnumerable<ModelValidatorProvider>
        GetModelValidatorProviders(this ServicesContainer services);
}

public class DefaultServices : ServicesContainer
{
    //其他成员
    public DefaultServices(HttpConfiguration configuration)
    {
        //其他操作
        this.SetMultiple<ModelBinderProvider>(new ModelBinderProvider[] {
            new TypeConverterModelBinderProvider(),
            new TypeMatchModelBinderProvider(),
            new KeyValuePairModelBinderProvider(),
            new ComplexModelDtoModelBinderProvider(),
            new ArrayModelBinderProvider(),
            new DictionaryModelBinderProvider(),
            new CollectionModelBinderProvider(),
            new MutableObjectModelBinderProvider() });
    }
}
```

`ModelBinderParameterBinding` 利用 `Model` 绑定机制来完成针对 `Action` 参数的绑定。如下面的代码片段所示, 我们在创建 `ModelBinderParameterBinding` 的时候需要指定一个 `ModelBinder` 对象, 它被用于初始化同名只读属性 `Binder`。当 `ModelBinderParameterBinding` 在进行参数绑定的时候, 会根据描述参数元数据的 `HttpParameterDescriptor` 对象创建一个表示当前 `Model` 上下文的 `ModelBindingContext` 对象, 然后将它作为参数调用 `ModelBinder` 的 `BindModel` 方法实施 `Model` 绑定。成功绑定后 `ModelBindingContext` 的 `Model` 属性值就是绑定的参数值。

```
public class ModelBinderParameterBinding : HttpParameterBinding, ...
{
    //其他成员
```

```

public ModelBinderParameterBinding(HttpParameterDescriptor descriptor,
    IModelBinder modelBinder,
    IEnumerable<ValueProviderFactory> valueProviderFactories);
public override Task ExecuteBindingAsync(
    ModelMetadataProvider metadataProvider, HttpContext actionContext,
    CancellationToken cancellationToken);

public IModelBinder Binder { get; }
}

```

我们可以在 Action 方法的某个参数上或者数据类型上应用 `ModelBinderAttribute` 特性来指定采用的 `ModelBinder`。如下面的代码片段所示, `ModelBinderAttribute` 继承自抽象类型 `System.Web.Http.ParameterBindingAttribute`, 后者可以应用在 Action 方法的某个参数或者参数类型上用以控制参数绑定采用的 `HttpParameterBinding` 对象。`ParameterBindingAttribute` 对 `HttpParameterBinding` 的提供实现在抽象方法 `GetBinding` 中。

```

public class ModelBinderAttribute : ParameterBindingAttribute
{
    public ModelBinderAttribute();
    public ModelBinderAttribute(Type binderType);
    public override HttpParameterBinding GetBinding(
        HttpParameterDescriptor parameter);

    public Type        BinderType { get; set; }
    public string      Name { get; set; }
    public bool        SuppressPrefixCheck { get; set; }
}

[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Class,
    Inherited=true, AllowMultiple=false)]
public abstract class ParameterBindingAttribute : Attribute
{
    public abstract HttpParameterBinding GetBinding(
        HttpParameterDescriptor parameter);
}

```

`ModelBinderAttribute` 通过实现 `GetBinding` 方法来提供对应的 `ModelBinderParameterBinding` 对象。我们可以通过构造函数的参数 `binderType` 或者 `BinderType` 属性来指定 `ModelBinder` 或者是 `ModelBinderProvider` 的类型。

`ModelBinder` 在默认情况下会根据参数名称来提取相应的数据, 如果提供的数据项具有不同于参数名称的前缀, 数据源与参数的匹配将会失败。在这种情况下我们需要在参数上应用 `ModelBinderAttribute` 属性并将此前缀赋值给 `Name` 属性, `HttpParameterDescriptor` 的 `Prefix` 属性值来源于此。另一个属性 `SuppressPrefixCheck` 表示在进行 `Model` 绑定的时候是否需要考虑指定的前缀。

`ModelBinderAttribute` 定义了一个默认无参的构造函数, 这意味着应用在参数或者数据类型上的 `ModelBinderAttribute` 特性可以不用指定具体的 `ModelBinder/ModelBinderProvider` 类型, 在这种情况下它会利用当前 `HttpConfiguration` 的 `ServicesContainer` 得到所有预先注册的 `ModelBinderProvider` 对象。如果获得的 `ModelBinderProvider` 列表中只有一个元素, 那么它直接被用来创建相应的 `ModelBinder`, 否则会创建一个具有如下定义的

CompositeModelBinderProvider 对象。

```
public sealed class CompositeModelBinderProvider : ModelBinderProvider
{
    //其他成员
    public CompositeModelBinderProvider();
    public CompositeModelBinderProvider(
        IEnumerable<ModelBinderProvider> providers);
    public override IModelBinder GetBinder(HttpConfiguration configuration,
        Type modelType);

    public IEnumerable<ModelBinderProvider> Providers { get; }
}
```

CompositeModelBinderProvider 是一个创建的“复合型” ModelBinderProvider，它通过提供的多个具体的 ModelBinderProvider 对象来创建一组 ModelBinder 对象，最终创建一个具有如下定义的 CompositeModelBinder 对象。CompositeModelBinder 是一个 ModelBinder 的组合，它本身并不真正实施 Model 绑定，具体的 Model 绑定工作交付给包含的 ModelBinder 列表来完成。

```
public class CompositeModelBinder : IModelBinder
{
    public CompositeModelBinder(IEnumerable<IModelBinder> binders);
    public CompositeModelBinder(params IModelBinder[] binders);
    public virtual bool BindModel(HttpContext actionContext,
        ModelBindingContext bindingContext);
}
```

最后一点值得一提的是，ModelBinderParameterBinding 在进行参数绑定的时候不会读取请求消息的主体内容，其数据主要来源于两个方面，即请求的查询字符串和通过 URL 路由得到的路由数据。

基于 HttpParameterDescriptor 的 HttpParameterBinding 创建

通过前面的介绍我们知道了针对目标 Action 方法的参数绑定是通过 HttpActionBinding 来完成的，而针对具体某个参数的绑定则是通过 HttpParameterBinding 来实现的。ASP.NET Web API 利用注册的 ActionValueBinder 来创建 HttpActionBinding，而默认使用的 ActionValueBinder 的类型为 DefaultActionValueBinder。在创建 HttpActionBinding 的时候，我们需要为目标 Action 方法的每个参数生成相应的 HttpParameterBinding 对象，现在着重讨论的是 DefaultActionValueBinder 如何根据用于描述参数的 HttpParameterDescriptor 对象来创建相应的 HttpParameterBinding。

在正式介绍针对 HttpParameterDescriptor 的 HttpParameterBinding 创建机制之前，我们需要先了解一个重要的特性 System.Web.Http.FromBodyAttribute。作为参数的数据主要具有三个来源，即路由数据、查询字符串和请求消息的主体内容，而 FromBodyAttribute 特性明确指明绑定的目标参数来源于请求消息的主体。从如下所示的 FromBodyAttribute 定义可以看出该特性可以同时应用到参数和类型上，意味着我们可以将它应用到 Action 方法的参数和数据类型上。

```
[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Class,
    Inherited = true, AllowMultiple = false)]
public sealed classFromBodyAttribute : Attribute
{
    public FromBodyAttribute();
}
```

图 9-15 揭示了创建 `HttpParameterBinding` 的整个流程。由于 `ModelBinderParameterBinding` 在进行针对 `Model` 绑定的参数绑定时并不会读取请求消息的主体内容，所以当我们在 `Action` 方法的某个参数上应用了 `ModelBinderAttribute` 特性之后，不可以在参数或者参数类型上再应用 `FromBodyAttribute` 特性，否则会抛出一个 `InvalidOperationException` 异常并提示特性冲突。实际上在这种情况下，`DefaultActionValueBinder` 会为该参数创建一个 `ErrorParameterBinding` 对象作为其对应的 `HttpParameterBinding`。

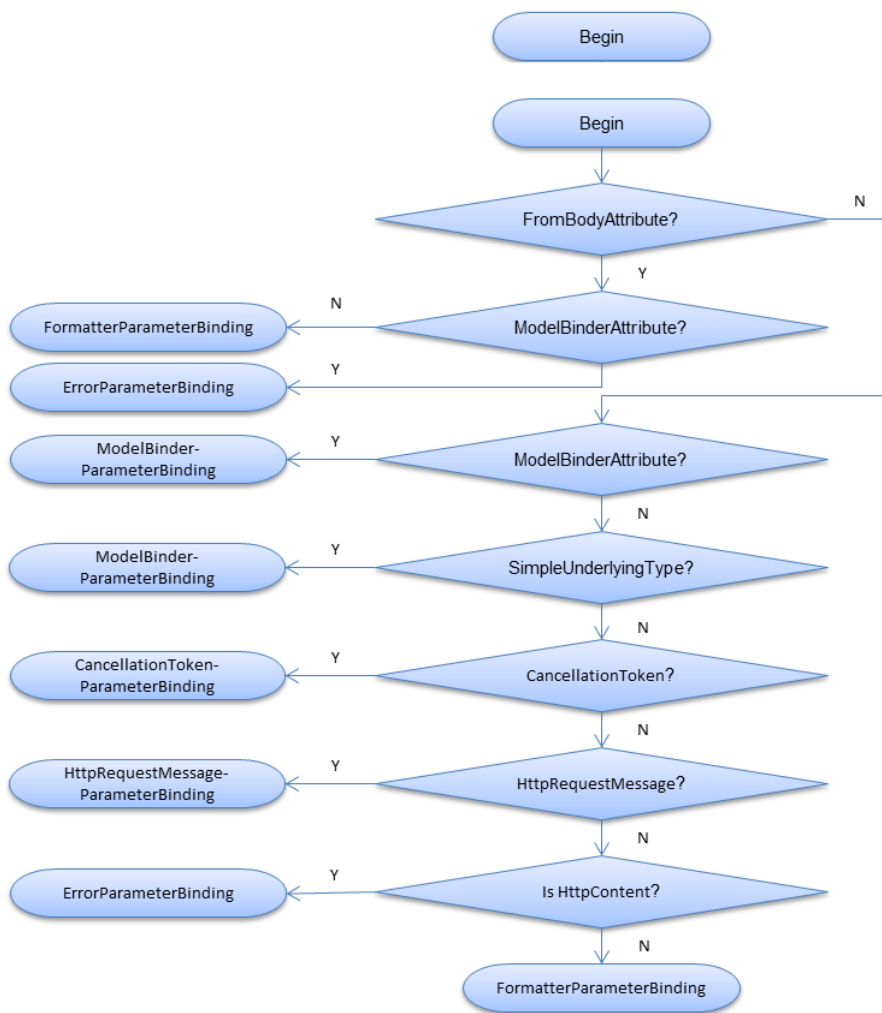


图 9-15 `HttpParameterBinding` 的创建流程

如果参数或参数类型上应用了 `FromBodyAttribute` 特性，并且没有应用 `ModelBinderAttribute` 特性，最终被创建的 `HttpParameterBinding` 是一个 `FormatterParameterBinding` 对象。反之，如果应用了 `ModelBinderAttribute` 特性，而没有应用 `FromBodyAttribute` 特性，`DefaultActionValueBinder` 最终会根据应用的 `ModelBinderAttribute` 来创建相应的 `ModelBinderParameterBinding` 对象。

在上述两个特性都不存在的情况下，创建的 `HttpParameterBinding` 类型完全取决于参数本身的类型。具体来说，如果参数本身类型或者它“基本类型（Underlying Type）”（针对可空值类型）是简单类型，`DefaultActionValueBinder` 会创建一个 `ModelBinderParameterBinding` 对象，这里的简单类型包括所有的原生类型（Primary Type）以及 `String`、`DateTime`、`Decimal`、`DateTimeOffset`、`GUID`，还包括所有支持与字符串类型进行转换的类型。

如果参数类型为 `CancellationToken` 或者 `HttpRequestMessage`，`DefaultActionValueBinder` 会分别创建 `CancellationTokenParameterBinding` 和 `HttpRequestMessageParameterBinding` 对象作为该参数对应的 `HttpParameterBinding`。`Action` 方法不应该包含类型为 `HttpContent` 的参数，这样的参数会采用一个 `ErrorParameterBinding` 对象作为其 `HttpParameterBinding`，意味着在进行参数绑定的时候会抛出异常。如果不满足前面这些条件，ASP.NET Web API 最终会创建一个 `FormatterParameterBinding` 来通过读取消息主体内容获取绑定参数的数据。

由于 `FormatterParameterBinding` 在进行参数绑定的时候是将整个请求消息的主体内容作为数据源，所以一个 `HttpActionBinding` 对象的 `ParameterBindings` 属性返回的 `HttpParameterBinding` 列表只允许包含一个 `FormatterParameterBinding` 对象。当 `DefaultActionValueBinder` 通过上述的流程为目标 `Action` 创建相应的 `HttpParameterBinding` 列表之后，会根据 `HttpParameterBinding` 的 `WillReadBody` 属性确保只有唯一一个能够读取请求消息的主体。如果发现多个 `WillReadBody` 属性为 `True` 的 `HttpParameterBinding`，它们都会被替换成 `ErrorParameterBinding`。

我们不妨通过一个简单的实例来演示 `DefaultActionValueBinder` 创建 `HttpParameterBinding` 的流程。在一个 ASP.NET MVC 应用中定义了如下一个继承自 `ApiController` 的 `FooController`，`Action` 方法 `Bar` 具有 8 个参数分别对应于图 9-15 所示流程的 8 个分支。

```
public class FooController : ApiController
{
    public void Bar(
        [ModelBinder]
        [FromBody]
        string parameter1,
        [ModelBinder]
        string parameter2,
        [FromBody]
        string parameter3,
        string parameter4,
        CancellationToken parameter5,
        HttpRequestMessage parameter6,
        HttpContent parameter7,
```

```

        Baz parameter8
    )
    { }
}

```

```

public class Baz
{ }

```

然后创建了如下一个 HomeController，在默认的 Action 方法 Index 中我们创建了一个 `HttpActionDescriptor`，它对应着定义在 `FooController` 中唯一的 Action 方法 `Bar`。然后我们利用当前 `HttpConfiguraiton` 得到注册的 `ActionValueBinder` 对象，并将创建的 `HttpActionDescriptor` 对象作为参数调用其 `GetBinding` 方法得到一个 `HttpActionBinding` 对象。最后我们将 `HttpActionBinding` 对象的属性 `ParameterBindings` 表示的 `HttpParameterBinding` 列表作为 Model 呈现在默认的 View 中。

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        HttpControllerDescriptor controllerDescriptor =
            new HttpControllerDescriptor {
                ControllerType = typeof(FooController) };
        MethodInfo methodInfo = typeof(FooController).GetMethod("Bar");
        HttpActionDescriptor actionDescriptor =
            new ReflectedHttpActionDescriptor(controllerDescriptor, methodInfo)
            { Configuration = GlobalConfiguration.Configuration };
        IActionValueBinder valueBinder = GlobalConfiguration.Configuration
            .Services.GetActionValueBinder();
        HttpActionBinding actionBinding =
            valueBinder.GetBinding(actionDescriptor);
        return View(actionBinding.ParameterBindings);
    }
}

```

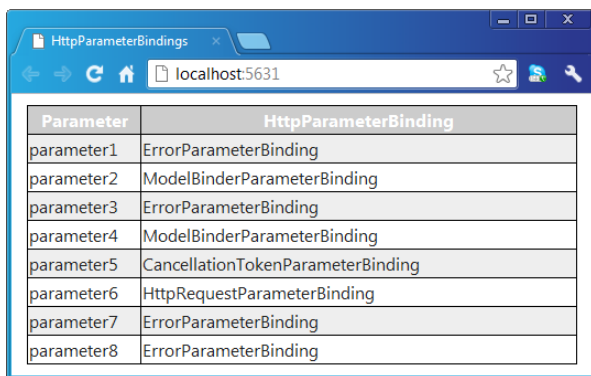
如下所示的是 Action 方法 Index 对应 View 的定义，这是一个将 `HttpParameterBinding` 集合作为 Model 的强类型 View。在该 View 中，我们每个 `HttpParameterBinding` 对应的参数名称和自身的类型以表格的形式呈现出来。

```

@model IEnumerable<HttpParameterBinding>
<html>
    <head>
        <title>HttpParameterBindings</title>
    </head>
    <body>
        <table>
            <tr><th>Parameter</th><th>HttpParameterBinding</th></tr>
            @foreach (var parameterBinding in Model)
            {
                <tr>
                    <td>@parameterBinding.Descriptor.ParameterName</td>
                    <td>@parameterBinding.GetType().Name</td>
                </tr>
            }
        </table>
    </body>
</html>

```

该程序运行之后会在浏览器中呈现出如图 9-16 所示的输出结果。仔细分析这个输出的列表我们会发现除了 parameter3 和 parameter8 之外，其余 6 个参数对应的 `HttpParameterBinding` 都与如图 9-15 所示的流程是吻合的。按照上述流程，parameter3 和 parameter8 本都应该是 `FormatterParameterBinding` 才对，由于它们的 `WillReadBody` 属性为 `True`，所以被替换成了 `ErrorParameterBinding`。如果我们注释掉任意一个参数，另一个参数对应的 `HttpParameterBinding` 将会变成 `FormatterParameterBinding`。（S910）



Parameter	HttpParameterBinding
parameter1	ErrorParameterBinding
parameter2	ModelBinderParameterBinding
parameter3	ErrorParameterBinding
parameter4	ModelBinderParameterBinding
parameter5	CancellationTokenParameterBinding
parameter6	HttpRequestParameterBinding
parameter7	ErrorParameterBinding
parameter8	ErrorParameterBinding

图 9-16 通过 `DefaultActionValueBinder` 创建的 `HttpParameterBinding` 类型

9.3.6 Model 验证

当 `HttpParameterBinding` 在生成参数对象之后会对其进行验证，不过这只会发生在 `FormatterParameterBinding` 中。其实这也很好理解，因为针对输入的验证在大部分情况下指的是针对自定义容器数据类型的验证。对于我们常用的几种 `HttpParameterBinding` 来说，这样的参数对象只能通过唯一能够读取请求消息主体内容的 `FormatterParameterBinding` 来绑定。

`FormatterParameterBinding` 利用一个名为 `BodyModelValidator` 的组件对绑定的参数对象实施验证，`BodyModelValidator` 实现了具有如下定义的 `System.Web.Http.Validation.IBodyModelValidator` 接口，它唯一的方法 `Validate` 对作为参数 `model` 的数据对象实施验证。该方法还包含其他的参数，`metadataProvider` 表示提供 `Model` 元数据的 `ModelMetadataProvider`，`actionContext` 表示当前 `HttpActionContext`，而 `keyPrefix` 表示 `Model` 名称前缀（对应 `HttpParameterDescriptor` 的 `Prefix` 属性）。

```
public class FormatterParameterBinding : HttpParameterBinding
{
    //其他成员
    public IBodyModelValidator BodyModelValidator { get; set; }
}

public interface IBodyModelValidator
{
```

```

    bool Validate(object model, Type type,
        ModelMetadataProvider metadataProvider, HttpContext actionContext,
        string keyPrefix);
}

```

和前面介绍的大部分组件一样, `BodyModelValidator` 依然是注册在当前 `HttpConfiguration` 的 `ServicesContainer` 中, 可以调用 `ServicesContainer` 具有如下定义的扩展方法 `GetBodyModelValidator` 获取注册的 `BodyModelValidator` 对象。默认注册的 `BodyModelValidator` 类型为具有如下定义的 `System.Web.Http.Validation.DefaultBodyModelValidator`。

```

public static class ServicesExtensions
{
    //其他成员
    public static IBodyModelValidator GetBodyModelValidator(
        this ServicesContainer services);
}

public class DefaultBodyModelValidator : IBodyModelValidator
{
    //其他成员
    public bool Validate(object model, Type type,
        ModelMetadataProvider metadataProvider, HttpContext actionContext,
        string keyPrefix);
}

```

实现在 `DefaultBodyModelValidator` 中的 Model 验证机制与 ASP.NET MVC 类似, 在这里我们仅仅作概括性的介绍。Model 验证最终通过组件 `ModelValidator` 来完成, 它继承自具有如下定义的抽象类 `System.Web.Http.Validation.ModelValidator`。Model 验证通过调用它的 `Validate` 方法来完成, 验证的结果通过以 `System.Web.Http.Validation.ModelValidationResult` 列表的形式返回。如下面的代码片段所示, `ModelValidationResult` 仅仅具有 `MemberName` 和 `Message` 两个属性, 分别表示数据成员名称和验证错误消息。`IsRequired` 属性表示目标数据成员是否是必需的, 这里返回 `False`。

```

public abstract class ModelValidator
{
    //其他成员
    protected ModelValidator(
        IEnumerable<ModelValidatorProvider> validatorProviders);
    public abstract IEnumerable<ModelValidationResult> Validate(
        ModelMetadata metadata, object container);

    public virtual bool IsRequired { get; }
    protected IEnumerable<ModelValidatorProvider> ValidatorProviders
        { get; }
}

public class ModelValidationResult
{
    public string MemberName { get; set; }
    public string Message { get; set; }
}

```


`ModelValidator` 类型具有一个受保护的属性 `ValidatorProviders`，它返回一个用于提供具体 `ModelValidator` 的 `ModelValidatorProvider` 列表，该属性在构造函数中通过 `validatorProviders` 参数指定。`ModelValidatorProvider` 继承自如下的抽象类 `System.Web.Http.Validation.ModelValidatorProvider` 类型，它的 `GetValidators` 返回提供的 `ModelValidator` 列表。

```
public abstract class ModelValidatorProvider
{
    public abstract IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata,
        IEnumerable<ModelValidatorProvider> validatorProviders);
}
```

ASP.NET Web API 在 `System.Web.Http.Validation.Validators` 命名空间下为我们定义了如下 4 种常用的 `ModelValidator` 类型。

- **DataAnnotationsModelValidator**：根据应用在数据类型或者属性上的验证特性创建的 `ModelValidator`。
- **RequiredMemberModelValidator**：专门用于对必需数据成员实施验证的 `ModelValidator`。
- **ValidatableObjectAdapter**：专门用于对实现了 `IValidatableObject` 接口的数据类型实施验证的 `ModelValidator`。
- **ErrorModelValidator**：它所谓的验证逻辑很简单，即仅仅直接抛出一个 `InvalidOperationException` 异常，异常的消息在 `ErrorModelValidator` 初始化的时候指定。

`DataAnnotationsModelValidator` 和 `ValidatableObjectAdapter` 提供了两种基本的 `Model` 验证方式，所以我们既可以通过应用验证特性来定义验证规则，也可以让数据类型实现 `IValidatableObject` 接口。`RequiredMemberModelValidator` 重写了 `IsRequired` 属性，使之返回 `True`。

我们常用的 `ModelValidatorProvider` 主要包括如下几种类型，它们均定义在命名空间 `System.Web.Http.Validation.Providers` 下。

- **DataAnnotationsModelValidatorProvider**：由它提供的 `ModelValidator` 列表中，包括基于应用在当前数据类型或者属性成员上所有验证特性创建的 `DataAnnotationsModelValidator`。如果数据类型实现了 `IValidatableObject` 接口，提供的 `ModelValidator` 还包括根据传入 `ModelValidatorProvider` 列表创建的 `ValidatableObjectAdapter` 对象。
- **DataMemberModelValidatorProvider**：数据类型还可以按照 WCF 的方式通过 `DataContractAttribute` 和 `DataMemberAttribute` 特性定义成“数据契约”。对于针对某个属性的数据成员，如果应用的 `DataMemberAttribute` 特性的 `IsRequired` 属性被设置成 `True`，意味着这是一个必需的数据成员，这与在属性成员上应用 `RequiredAttribute` 特性具有相同的效果。在这种情况下 `DataMemberModelValidatorProvider` 会提供一个包含一个

RequiredMemberModelValidator 对象的 ModelValidator 列表。

- **RequiredMemberModelValidatorProvider**：它用于提供对数据成员作必要性验证的 RequiredMemberModelValidator。当我们调用 FormDataCollection 的扩展方法 ReadAs<T> 读取表单数据并反序列化成指定类型的数据对象的时候，会利用它来创建对应的 RequiredMemberModelValidator 对必需的数据成员实施验证。
- **InvalidModelValidatorProvider**：它用于提供 ErrorModelValidator 来提示应用在目标数据类型上相应的验证特性不合法。具体来说不合法的验证特性应用方法包括：将验证特性应用在字段或者非公有实例属性上；将 RequiredAttribute 和 DataMemberAttribute 应用在同一个数据成员上，但是 DataMemberAttribute 的 IsRequired 属性为 False。

当前使用的所有 ModelValidatorProvider 都实现注册在 HttpConfiguraiton 的 ServicesContainer 中，我们可以调用 ServicesContainer 如下一个扩展方法 GetModelValidatorProviders 得到所有注册的 ModelValidatorProvider。如下面的代码片段所示，在 DefaultServices 初始化的时候会注册三个具体的 ModelValidatorProvider 对象，它们的类型分别是 DataAnnotationsModelValidatorProvider、DataMemberModelValidatorProvider 和 InvalidModelValidatorProvider。

```
public static class ServicesExtensions
{
    //其他成员
    public static IEnumerable<ModelValidatorProvider>
        GetModelValidatorProviders(this ServicesContainer services);
}

public class DefaultServices : ServicesContainer
{
    //其他成员
    public DefaultServices(HttpConfiguration configuration)
    {
        this.SetMultiple<ModelValidatorProvider>(
            new ModelValidatorProvider[] {
                new DataAnnotationsModelValidatorProvider(),
                new DataMemberModelValidatorProvider(),
                new InvalidModelValidatorProvider() });
    }
}
```

ASP.NET MVC 的 Model 验证是伴随着 Model 绑定一起向前推进的，但是 ASP.NET Web API 中的 Model 验证则是在 FormatterParameterBinding 成功得到了参数对象之后再对其进行验证，所以 Model 验证本身在 ASP.NET MVC 中并非递归进行的（表现出来的递归是源于递归进行的 Model 绑定），但是 Model 验证在 ASP.NET Web API 中则是递归地进行的。具体的验证逻辑其实很简单，它通过注册的 ModelValidatorProvider 根据 Model 元数据创建相应的 ModelValidator 对数据对象实施验证，如果被验证数据类型是一个集合或者复杂类型，则按照相同的方式递归地验证所有集合元素或者属性成员。

对于 ASP.NET MVC 的 Model 验证来说, 验证失败后结果以 `ModelError` 的形式写入当前的 `ModelState` 中, 这一点与 ASP.NET Web API 是一致的。ASP.NET Web API 同样定义 `ModelStateDictionary`、`ModelState`、`ModelError` 等一系列类型, 不过它们都存在于 `System.Web.Http.ModelBinding` 命名空间下。如下面的代码片段所示, `ApiController` 的 `ModelState` 属性返回的正是 `ModelStateDictionary` 对象, 我们可以通过它来确定输入数据是否通过验证, 也可以获取所有的验证错误消息。

```
public abstract class ApiController : IHttpController, IDisposable
{
    //其他成员
    public ModelStateDictionary ModelState { get; }
}
```

ASP.NET Web API 下的 Model 验证的最后的步骤就是将验证结果写入当前 `ModelState` 中, 其本身并不会抛出异常, 也不会自动被写入当前的响应, 我们需要自行处理验证的结果, 可以通过自定义 `ActionFilter` 的方式来自动将验证错误响应给客户端。

9.3.7 Action 的执行与结果的响应

接下来我们来讨论 `HttpControllerHandler` 请求处理流程的最后一个环节, 即如何执行被激活 `HttpController` 的 `Action` 方法, 以及如何将执行的结果响应给客户端。ASP.NET Web API 针对目标 `Action` 方法的执行通过一个叫做 `HttpActionInvoker` 的组件来完成。

HttpActionInvoker

`HttpActionInvoker` 实现了具有如下定义的 `System.Web.Http.Controllers.IHttpActionInvoker` 接口, 目标 `Action` 的执行通过调用 `InvokeActionAsync` 方法来完成。用于描述 `HttpController` 的 `HttpControllerDescriptor` 具有如下所示的 `HttpActionInvoker` 属性, 它最终完成了对目标 `Action` 的执行。

```
public interface IHttpActionInvoker
{
    Task<HttpResponseMessage> InvokeActionAsync(HttpActionContext actionContext,
        CancellationToken cancellationToken);
}

public class HttpControllerDescriptor
{
    //其他成员
    public IHttpActionInvoker HttpActionInvoker { get; set; }
}
```

`HttpActionInvoker` 依然是注册在当前 `HttpConfiguration` 的 `ServicesContainer` 中, 我们可以调用 `ServicesContainer` 的如下一个扩展方法 `GetActionInvoker` 来得到注册的 `HttpActionInvoker` 对象。默认注册的 `HttpActionInvoker` 类型为 `System.Web.Http.Controllers.`

`ApiControllerActionInvoker`，其定义如下。

```
public static class ServicesExtensions
{
    //其他成员
    public static IHttpActionInvoker GetActionInvoker(
        this ServicesContainer services);
}

public class ApiControllerActionInvoker : IHttpActionInvoker
{
    public virtual Task<HttpResponseMessage> InvokeActionAsync(
        HttpContext actionContext,
        CancellationToken cancellationToken);
}
```

`ApiControllerActionInvoker` 具有两个核心功能，一个是执行目标 `Action` 方法，另一个则是将执行的结果转换成 `HttpResponseMessage` 对象。第一个功能是通过调用描述目标 `Action` 的 `HttpActionDescriptor` 对象的 `ExecuteAsync` 来完成的。如下面的代码片段所示，这个方法具有两个参数，前者为当前 `HttpContext`，另一个是通过 `HttpActionBinding` 生成的参数列表。默认的 `ReflectedHttpActionDescriptor` 通过它封装的 `MethodInfo` 以反射的方式完成了对 `Action` 方法的执行。

```
public abstract class HttpActionDescriptor
{
    //其他成员
    public abstract Task<object> ExecuteAsync(
        HttpContext controllerContext,
        IDictionary<string, object> arguments);
}

public class ReflectedHttpActionDescriptor : HttpActionDescriptor
{
    //其他成员
    public MethodInfo MethodInfo { get; set; }
}
```

现在我们关注 `ApiControllerActionInvoker` 实现的另一个核心功能，看看它是如何将 `Action` 方法执行的结果转换成一个表示响应的 `HttpResponseMessage` 对象的，这涉及到另一个被称为 `ActionResultConverter` 的组件。

ActionResultConverter

旨在将执行 `Action` 方法的返回值转换成 `HttpResponseMessage` 的 `ActionResultConverter` 实现了具有如下定义的接口 `System.Web.Http.Controllers.IActionResultConverter`。具体的转换工作通过唯一的方法 `Convert` 实现，该方法的第二个参数代表执行 `Action` 方法的返回值。`HttpActionDescriptor` 的属性 `ResultConverter` 返回的 `ActionResultConverter` 被真正用于完成这样的转换工作。

```

public interface IActionResultConverter
{
    HttpResponseMessage Convert(HttpControllerContext controllerContext,
        object actionResult);
}

public abstract class HttpActionDescriptor
{
    //其他成员
    public virtual IActionResultConverter ResultConverter { get; }
}

```

针对 Action 方法返回类型的不同，ASP.NET Web API 定义了如下三种类型的 IActionResultConverter。

- **ResponseMessageResultConverter**：它在 Action 方法的返回类型为 HttpResponseMessage 或者其子类的情况下被使用，这种情况下无需作任何转化。
- **ValueResultConverter<T>**：它在 Action 方法返回一个具体的对象时被使用。在这种情况下会根据请求期望的媒体类型选择对应的 MediaTypeFormatter 对返回对象进行序列化，并根据序列化后的内容生成一个状态为“200， Succeed”的 HttpResponseMessage。
- **VoidResultConverter**：它用在 Action 方法不具有返回值（或者说返回类型为 void）的情况。由于没有具体响应的内容，它仅仅创建一个空的 HttpResponseMessage 而已。

当目前为止，HttpControllerDispatcher 完成了 Action 方法的执行并将执行的结果转换成一个 HttpResponseMessage 对象。接下来它会将 HttpResponseMessage 沿着 HttpResponseMessageHandler 管道相反的方向传递，每一个具体的 HttpResponseMessageHandler 都可以对它作相应的处理。HttpResponseMessage 最后被传递到 HttpControllerHandler 这个自定义的 HttpHandler 中，由它对请求作最终的响应。

筛选器的执行

我们可以按照 ASP.NET MVC 一样的方式在 ASP.NET Web API 中使用筛选器，由于 ASP.NET Web API 没有 ActionResult 的概念，所以只支持 AuthorizationFilter、ActionFilter 和 ExceptionFilter。这三种筛选器分别实现了具有如下定义的三个接口 IAuthorizationFilter、IActionFilter 和 IExceptionFilter，它们都定义在 System.Web.Http.Filters 命名空间下。

```

public interface IAuthorizationFilter : IFilter
{
    Task<HttpResponseMessage> ExecuteAuthorizationFilterAsync(
        HttpActionContext actionContext, CancellationToken cancellationToken,
        Func<Task<HttpResponseMessage>> continuation);
}

public interface IActionFilter : IFilter
{
    Task<HttpResponseMessage> ExecuteActionFilterAsync(
        HttpActionContext actionContext, CancellationToken cancellationToken,
        Func<Task<HttpResponseMessage>> continuation);
}

```

```

}

public interface IExceptionHandler : IFilter
{
    Task ExecuteExceptionHandlerAsync(
        HttpActionExecutedContext actionExecutedContext,
        CancellationToken cancellationToken);
}

public interface IFilter
{
    bool AllowMultiple { get; }
}

```

除了这三个接口，ASP.NET Web API 还定义了三个实现了这些接口的抽象类型，它们分别是具有如下定义的 `AuthorizationFilterAttribute`、`ActionFilterAttribute` 和 `ExceptionHandlerAttribute`，我们自定义的筛选器特性一般将它们作为基类。

```

[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
    Inherited=true, AllowMultiple=true)]
public abstract class AuthorizationFilterAttribute : FilterAttribute,
    IAuthorizationFilter, IFilter
{
    protected AuthorizationFilterAttribute();
    public virtual void OnAuthorization(HttpActionContext actionContext);
    Task<HttpResponseMessage>
IAuthorizationFilter.ExecuteAuthorizationFilterAsync(
    HttpActionContext actionContext, CancellationToken cancellationToken,
    Func<Task<HttpResponseMessage>> continuation);
}

[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
    Inherited=true, AllowMultiple=true)]
public abstract class ActionFilterAttribute : FilterAttribute,
    IActionFilter, IFilter
{
    protected ActionFilterAttribute();
    public virtual void OnActionExecuted(
        HttpActionExecutedContext actionExecutedContext);
    public virtual void OnActionExecuting(HttpActionContext actionContext);
    Task<HttpResponseMessage> IActionFilter.ExecuteActionFilterAsync(
        HttpActionContext actionContext, CancellationToken cancellationToken,
        Func<Task<HttpResponseMessage>> continuation);
}

[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
    Inherited=true, AllowMultiple=true)]
public abstract class ExceptionFilterAttribute : FilterAttribute,
    IExceptionHandler, IFilter
{
    protected ExceptionFilterAttribute();
    public virtual void OnException(
        HttpActionExecutedContext actionExecutedContext);
    Task IExceptionHandler.ExecuteExceptionHandlerAsync(
        HttpActionExecutedContext actionExecutedContext,
        CancellationToken cancellationToken);
}

```

三种不同类型的筛选器在 ASP.NET Web API 的作用和执行方式与 ASP.NET MVC 基本一致，所以在这里就不对它们进行深入介绍了。

9.4 Web API 的调用和自我寄宿

Web API 的调用本质上就是一个单纯的 HTTP 请求/响应的过程，客户端只需向目标 URL 发送相应的请求就能够调用期望的服务，并通过响应得到服务操作执行的结果，所以完全可以按照我们熟悉的网络编程（比如直接使用 `HttpWebRequest/HttpWebResponse`）实现请求的发送和响应解析。在 Web 应用中，我们还经常采用 Ajax 的方式来进行 Web API 的调用，为了方便在托管程序中的调用，ASP.NET Web API 为我们定义了一个 `System.Net.Http.HttpClient` 的类型。

9.4.1 HttpClient

我们在前面已经演示过针对 `HttpClient` 的 Web API 调用，在这里对它进行一个相对系统的介绍。`HttpClient` 实际上继承自具有如下定义的抽象类 `System.Net.Http.HttpMessageInvoker`，请求的发送和响应的接收实现在核心方法 `SendAsync` 中。在这里请求和响应消息通过我们熟悉的 `HttpRequestMessage` 和 `HttpResponseMessage` 表示。

```
public class HttpMessageInvoker : IDisposable
{
    public HttpMessageInvoker(HttpMessageHandler handler);
    public HttpMessageInvoker(HttpMessageHandler handler, bool disposeHandler);

    public void Dispose();
    public virtual Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken);
}
```

`HttpMessageInvoker` 可以看成是对一个 `HttpMessageHandler` 对象的封装，这个被封装的 `HttpMessageHandler` 通过构造函数的 `handler` 参数指定，构造函数的另一个参数 `disposeHandler` 表示这个被封装的 `HttpMessageHandler` 是否应该在 `HttpMessageInvoker` 被释放的时候随之被释放。这个被封装的 `HttpMessageInvoker` 最终帮助 `HttpClient` 完成了发送请求和接受响应的工作。

如下面的代码片段所示，继承自 `HttpMessageInvoker` 的 `HttpClient` 除了定义与基类匹配的构造函数外，还定义了一个无参默认的构造函数。`HttpClient` 定义了四个重载的 `SendAsync` 方法，其中参数 `completionOption` 类型为 `System.Net.Http.HttpCompletionOption` 枚举，用以提供判断操作完成的标志，两个枚举项 `ResponseHeadersRead` 和 `ResponseContentRead`（默认值）分别表示操作结束的标志是应该在读取完可用响应报头之后，还是在读取包括主体内容在内的整个响应消息之后。

```

public class HttpClient : HttpResponseMessageInvoker
{
    //其他成员
    public HttpClient();
    public HttpClient(HttpMessageHandler handler);
    public HttpClient(HttpMessageHandler handler, bool disposeHandler);

    public Task<HttpResponseMessage> SendAsync(HttpRequestMessage request);
    public Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
        HttpCompletionOption completionOption);
    public override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken);
    public Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
        HttpCompletionOption completionOption,
        CancellationToken cancellationToken);

    public Uri                BaseAddress { get; set; }
    public HttpRequestHeaders DefaultRequestHeaders { get; }
    public long                MaxResponseContentBufferSize { get; set; }
    public TimeSpan            Timeout { get; set; }
}

public enum HttpCompletionOption
{
    ResponseContentRead,
    ResponseHeadersRead
}

```

`HttpClient` 具有四个属性，其中 `BaseAddress` 用于指定 Web API 基地址；我们可以利用 `DefaultRequestHeaders` 属性为请求添加任意的 HTTP 报头；`MaxResponseContentBufferSize` 属性表示读取响应缓冲区允许的最大字节数，默认为 2G 字节；`Timeout` 属性则表示请求超时时限，默认值为 100 秒。

如果调用默认构造函数来创建 `HttpClient` 对象，ASP.NET Web API 会默认创建一个具有如下定义的 `System.Net.Http.HttpClientHandler` 对象作为其内部封装的 `HttpMessageHandler`。`HttpClientHandler` 的 `SendAsync` 方法实际上就是将 `HttpRequestMessage` 对象转换成真正的 HTTP 请求并发送出去，最后将接收的 HTTP 响应转换成 `HttpResponseMessage` 的过程。`HttpClient` 其实还定义了一系列属性用于控制请求发送和响应接收的行为，篇幅所限就不再对它们作详细介绍了。

```

public class HttpClientHandler : HttpMessageHandler
{
    //其他成员
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken);
}

```

`HttpClient` 直接利用内部封装的 `HttpMessageHandler` 实现请求的发送和响应的接收，如果我们需要在请求发送之前或者响应接收之后分别对 `HttpRequestMessage` 和 `HttpResponseMessage` 进行相应的处理，可以利用 `DelegatingHandler` 将这个内部 `HttpMessageHandler` 构建为一个“`HttpMessageHandler` 链”。客户端针对 Web API 的调用依然

具有一个类似于服务端的管道。如图 9-17 所示，HttpClient 和 HttpClientHandler 分别作为这个管道的首尾，中间是一个可任意组合的 HttpResponseMessage 链。

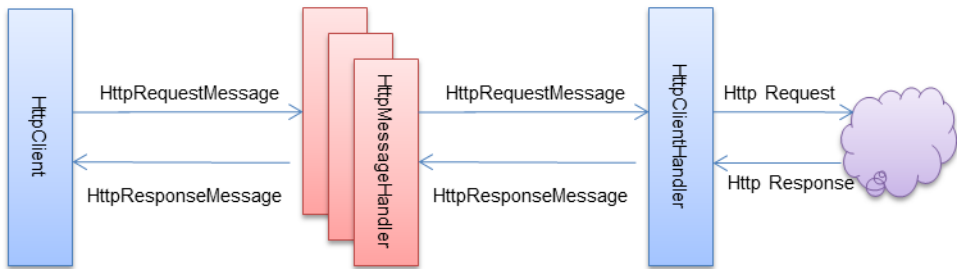


图 9-17 通过 HttpClient 构建的客户端管道

HttpClient 定义了一系列以 Get 作为前缀的方法用于向指定的 URL 发送 HTTP-GET 请求，并获取相应的响应内容，而请求的地址既可以采用 Uri 对象也可以使用字符串来表示。在这些方法中，GetAsync 返回的是一个 Task<HttpResponseMessage> 对象，GetByteArrayAsync、GetStreamAsync 和 GetStringAsync 则分别返回一个 Task<byte[]>、Task<Stream>和 Task<string>对象。我们可以调用泛型的 GetContentAsync<T>返回基于指定类型的 Task 对象，不过这需要在方法中指定用于读取响应内容并生成指定类型对象的 Func<HttpContent, Task<T>>委托。这些方法最终都会调用相应 SendAsync 方法。

```
public class HttpClient : HttpMessageInvoker
{
    //其他成员
    public Task<HttpResponseMessage> GetAsync(string requestUri);
    public Task<HttpResponseMessage> GetAsync(Uri requestUri);
    public Task<HttpResponseMessage> GetAsync(string requestUri,
        HttpCompletionOption completionOption);
    public Task<HttpResponseMessage> GetAsync(string requestUri,
        CancellationToken cancellationToken);
    public Task<HttpResponseMessage> GetAsync(Uri requestUri,
        HttpCompletionOption completionOption);
    public Task<HttpResponseMessage> GetAsync(Uri requestUri,
        CancellationToken cancellationToken);
    public Task<HttpResponseMessage> GetAsync(string requestUri,
        HttpCompletionOption completionOption,
        CancellationToken cancellationToken);
    public Task<HttpResponseMessage> GetAsync(Uri requestUri,
        HttpCompletionOption completionOption,
        CancellationToken cancellationToken);

    public Task<byte[]> GetByteArrayAsync(string requestUri);
    public Task<byte[]> GetByteArrayAsync(Uri requestUri);
    public Task<Stream> GetStreamAsync(string requestUri);
    public Task<Stream> GetStreamAsync(Uri requestUri);
    public Task<string> GetStringAsync(string requestUri);
    public Task<string> GetStringAsync(Uri requestUri);

    private Task<T> GetContentAsync<T>(Uri requestUri,
        HttpCompletionOption completionOption, T defaultValue,
```

```
Func<HttpContent, Task<T>> readAs);
}
```

对于 `GetByteArrayAsync`、`GetStringAsync` 和 `GetStreamAsync` 这三个方法来说，它们先通过调用 `SendAsync` 方法得到表示响应消息的 `HttpResponseMessage` 对象，然后通过其 `Content` 属性得到表示响应内容的 `HttpContent` 对象，最终调用 `HttpContent` 如下的三个方法得到结果类型分别为字节数组、`Stream` 和字符串的 `Task` 对象。

```
public class public abstract class HttpContent : IDisposable
{
    //其他成员
    public Task<byte[]> ReadAsByteArrayAsync();
    public Task<Stream> ReadAsStreamAsync();
    public Task<string> ReadAsStringAsync();
}
```

很多情况下我们需要直接将响应的内容（XML 或者 JSON 字符串）反序列化成指定类型的对象，在这种情况下可以先得到表示响应内容的 `HttpContent` 对象，并调用针对它如下所示的扩展方法 `ReadAsAsync<T>`，在调用 `ReadAsAsync<T>` 方法的时候需要指定一个 `MediaTypeFormatter` 列表。

```
public static class HttpContentExtensions
{
    //其他成员
    public static Task<T> ReadAsAsync<T>(this HttpContent content);
    public static Task<T> ReadAsAsync<T>(this HttpContent content,
        IEnumerable<MediaTypeFormatter> formatters);
    public static Task<T> ReadAsAsync<T>(this HttpContent content,
        IEnumerable<MediaTypeFormatter> formatters,
        IFormatterLogger formatterLogger);
}
```

在读取响应内容后，这些方法会根据响应的媒体类型在指定的列表中选择相应的 `MediaTypeFormatter` 对象，然后针对指定的泛型类型进行反序列化。如果没有显式指定 `MediaTypeFormatter` 列表，默认使用的 `MediaTypeFormatter` 列表会包含三个对象，它们的类型分别为 `JsonMediaTypeFormatter`、`XmlMediaTypeFormatter` 和 `FormUrlEncodedMediaTypeFormatter`。

GET、POST、PUT 和 DELETE 是我们进行 Web API 调用经常采用的四种 HTTP 方法，所以除了上面介绍的这些针对 HTTP-GET 的方法之外，`HttpClient` 还为我们定义了如下几个针对其他三个 HTTP 方法的方法。

```
public class HttpClient : HttpMessageInvoker
{
    //其他成员
    public Task<HttpResponseMessage> PostAsync(string requestUri,
        HttpContent content);
    public Task<HttpResponseMessage> PostAsync(Uri requestUri,
        HttpContent content);
    public Task<HttpResponseMessage> PostAsync(string requestUri,
        HttpContent content, CancellationToken cancellationToken);
    public Task<HttpResponseMessage> PostAsync(Uri requestUri,
        HttpContent content, CancellationToken cancellationToken);
}
```

```

public Task<HttpResponseBody> PutAsync(string requestUri,
    HttpContent content);
public Task<HttpResponseBody> PutAsync(Uri requestUri,
    HttpContent content);
public Task<HttpResponseBody> PutAsync(string requestUri,
    HttpContent content, CancellationToken cancellationToken);
public Task<HttpResponseBody> PutAsync(Uri requestUri,
    HttpContent content, CancellationToken cancellationToken);

public Task<HttpResponseBody> DeleteAsync(string requestUri);
public Task<HttpResponseBody> DeleteAsync(Uri requestUri);
public Task<HttpResponseBody> DeleteAsync(string requestUri,
    CancellationToken cancellationToken);
public Task<HttpResponseBody> DeleteAsync(Uri requestUri,
    CancellationToken cancellationToken);
}

```

对于 `PostAsync` 和 `PutAsync` 方法，请求的内容和与内容相关的 HTTP 报头只能通过 `HttpContent` 对象来指定，其实我们更希望的方式是能够直接指定一个对象作为请求的内容，这可以通过调用 `HttpClient` 如下所示的扩展方法来实现。其中 `PostAsJsonAsync<T>/PutAsJsonAsync<T>` 和 `PostAsXmlAsync<T>/PutAsXmlAsync<T>` 方法分别采用 `JsonMediaTypeFormatter` 和 `XmlMediaTypeFormatter` 对传入的对象进行序列化和相关内容报头的设置，而 `PostAsync<T>` 和 `PutAsJsonAsync<T>` 中的这两项操作则通过指定的 `MediaTypeFormatter` 来完成。

```

public static class HttpClientExtensions
{
    //PostAsJsonAsync<T>()
    public static Task<HttpResponseBody> PostAsJsonAsync<T>(
        this HttpClient client, string requestUri, T value);
    public static Task<HttpResponseBody> PostAsJsonAsync<T>(
        this HttpClient client, string requestUri, T value,
        CancellationToken cancellationToken);

    //PostAsXmlAsync<T>()
    public static Task<HttpResponseBody> PostAsXmlAsync<T>(
        this HttpClient client, string requestUri, T value);
    public static Task<HttpResponseBody> PostAsXmlAsync<T>(
        this HttpClient client, string requestUri, T value,
        CancellationToken cancellationToken);

    //PostAsync<T>()
    public static Task<HttpResponseBody> PostAsync<T>(
        this HttpClient client, string requestUri, T value,
        MediaTypeFormatter formatter);
    public static Task<HttpResponseBody> PostAsync<T>(
        this HttpClient client, string requestUri, T value,
        MediaTypeFormatter formatter, string mediaType);
    public static Task<HttpResponseBody> PostAsync<T>(
        this HttpClient client, string requestUri, T value,
        MediaTypeFormatter formatter, CancellationToken cancellationToken);
    public static Task<HttpResponseBody> PostAsync<T>(
        this HttpClient client, string requestUri, T value,
        MediaTypeFormatter formatter, string mediaType,
        CancellationToken cancellationToken);
}

```

```

//PutAsJsonAsync<T>()
public static Task<HttpResponseBody> PutAsJsonAsync<T>(
    this HttpClient client, string requestUri, T value);
public static Task<HttpResponseBody> PutAsJsonAsync<T>(
    this HttpClient client, string requestUri, T value,
    CancellationToken cancellationToken);

//PutAsXmlAsync <T>()
public static Task<HttpResponseBody> PutAsXmlAsync<T>(
    this HttpClient client, string requestUri, T value);
public static Task<HttpResponseBody> PutAsXmlAsync<T>(
    this HttpClient client, string requestUri, T value,
    CancellationToken cancellationToken);

//PutAsync<T>()
public static Task<HttpResponseBody> PutAsync<T>(
    this HttpClient client, string requestUri, T value,
    MediaTypeFormatter formatter);
public static Task<HttpResponseBody> PutAsync<T>(this HttpClient client,
    string requestUri, T value, MediaTypeFormatter formatter, string mediaType);
public static Task<HttpResponseBody> PutAsync<T>(this HttpClient client,
    string requestUri, T value, MediaTypeFormatter formatter,
    CancellationToken cancellationToken);
public static Task<HttpResponseBody> PutAsync<T>(this HttpClient client,
    string requestUri, T value, MediaTypeFormatter formatter,
    string mediaType, CancellationToken cancellationToken);
}

```

其实 ASP.NET Web API 还为 `HttpContent` 定义了一些扩展方法来读取 HTTP 请求/响应的内容，篇幅所限，我们就不一一列举了。

9.4.2 HttpSelfHostServer

与 WCF 服务类似，我们不仅仅可以创建一个 Web 应用作为 ASP.NET Web API 的宿主，也可以建立一个任意类型的托管应用（比如控制台应用、GUI 应用甚至 Windows 服务）来寄宿 ASP.NET Web API，我们将后者称为自我寄宿（Self Hosting）。ASP.NET Web API 的自我寄宿利用具有如下定义的 `System.Web.Http.SelfHost.HttpSelfHostServer` 类型来完成。

```

public sealed class HttpSelfHostServer : HttpServer
{
    //其他成员
    public HttpSelfHostServer(HttpSelfHostConfiguration configuration);
    public HttpSelfHostServer(HttpSelfHostConfiguration configuration,
        HttpMessageHandler dispatcher);

    public Task OpenAsync();
    public Task CloseAsync();
}

```

`HttpSelfHostServer` 实际上是 `HttpServer` 的子类。以 `HttpServer` 和 `HttpControllerDispatcher` 作为两个端点的 `HttpMessageHandler` 管道提供了一个请求处理、Controller 激活、Action 执行、响应生成与处理的通道，但是并不具备请求监听、接收以及最终对请求进行响应的能力，而这可以利用 `HttpSelfHostServer` 来实现。

如上面的代码片段所示，在构建 `HttpSelfHostServer` 对象的时候需要指定一个 `System.Web.Http.SelfHost.HttpSelfHostConfiguration` 类型的参数，`HttpSelfHostConfiguration` 是我们熟悉的 `HttpConfiguration` 的子类，它在基类的基础上为 Web API 的自我寄宿方式提供额外的配置信息。当我们调用 `HttpSelfHostServer` 的 `OpenAsync` 方法开启它之后，一个监听器被创建出来并绑定到指定的监听地址（通过 `HttpSelfHostConfiguration` 对象指定）进行请求的监听。

实际上 `HttpSelfHostServer` 完全采用了 WCF 的服务寄宿机制，这个监听器就是 `Binding` 对象创建的监听器。对 WCF 具有基本了解的人应该知道，终结点（Endpoint）是 WCF 最为核心的概念，而 `Binding` 是终结点的 ABC（Address、Binding 和 Contract）三要素之一，它实现了所有通信的细节。`Binding` 对监听器的创建，以及请求的接收、处理和响应是一个相对复杂的流程，我们不可能在这里对其进行深入探讨，对此有兴趣的读者可以参考笔者在 2012 年 4 月出版的《WCF 全面解析》（上、下册）。

`HttpSelfHostServer` 利用的是一个 `System.Web.Http.SelfHost.Channels.HttpBinding` 类型的 `Binding`。`Binding` 不仅仅创建一个用于监听请求的监听器，还会创建一个信道栈进行请求的接收和对响应的回复。监听器一旦探测到抵达的请求，会利用信道栈接收该请求。接下来它会利用 URL 路由机制对请求的 URL 进行解析并得到相应的路由数据，然后创建一个表示当前请求的 `HttpRequestMessage` 对象并递交给 `HttpMessageHandler` 管道进行处理。这个管道完成了我们熟悉的处理流程之后，表示响应消息的 `HttpResponseMessage` 对象通过 `HttpBinding` 的信道栈回复给客户端。

实例演示：针对控制台应用程序的 Web API 自我寄宿（S911）

在本章的最后我们来做一个基于 Web API 自我寄宿的实例演示，本实例中还会演示如何采用 `HttpClient` 进行 Web API 的调用。先利用 Visual Studio 创建一个空的解决方法，并在其中添加如图 9-18 所示的三个项目。类库项目 `Common` 用于定义数据类型，而控制台项目 `WebApi` 和 `Client` 用于定义寄宿 Web API 和模拟客户端程序，它们引用 `Common` 项目。

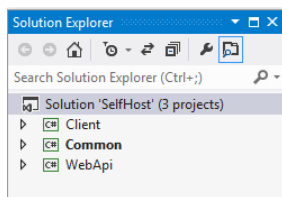


图 9-18 Web API 自我寄宿演示实例的解决方案结构

我们依然采用之前使用过的“联系人管理”的例子，为此我们先在项目 `Common` 中定义如下一个表示联系人的 `Contact` 数据类型。

```
public class Contact
{
    public string Id { get; set; }
```

```

public string Name { get; set; }
public string PhoneNo { get; set; }
public string EmailAddress { get; set; }
}

```

我们通过 Nuget 为 WebApi 项目添加针对针对 ASP.NET Web API 自我寄宿相关的引用，对应的 Nuget Package 为“Microsoft ASP.NET Web API Self Host (Microsoft.AspNet.WebApi.SelfHost)”，然后我们定义如下一个实现了简单 CRUD 操作的 ContactsController。

```

public class ContactsController : ApiController
{
    private static List<Contact> contacts = new List<Contact>
    {
        new Contact{ Id="001", Name = "张三", PhoneNo="123",
            EmailAddress="zhangsan@gmail.com"},
        new Contact{ Id="002", Name = "李四", PhoneNo="456",
            EmailAddress="lisi@gmail.com"}
    };

    public IEnumerable<Contact> Get()
    {
        return contacts;
    }

    public Contact Get(string id)
    {
        return contacts.FirstOrDefault(c => c.Id == id);
    }

    public void Put(Contact contact)
    {
        contacts.Add(contact);
    }

    public void Post(Contact contact)
    {
        Delete(contact.Id);
        contacts.Add(contact);
    }

    public void Delete(string id)
    {
        Contact contact = contacts.FirstOrDefault(c => c.Id == id);
        contacts.Remove(contact);
    }
}

```

对 ContactsController 的自我寄宿定义在默认创建的 Program 类型中。如下面的代码片段所示，在程序入口的 Main 方法中，我们针对监听地址 `http://127.0.0.1:3721` 创建了一个 `HttpSelfHostConfiguration` 对象，并进行相应的路由注册，然后我们针对它创建了一个 `HttpSelfHostServer` 对象并调用 `OpenAsync` 方法启动寄宿的 Web API。

```

class Program
{
    static void Main(string[] args)
    {

```

```

    HttpSelfHostConfiguration configuration =
        new HttpSelfHostConfiguration("http://127.0.0.1:3721");
    configuration.Routes.MapHttpRoute(
        "API Default",
        "api/{controller}/{id}",
        new { id = RouteParameter.Optional });

    using (HttpSelfHostServer server = new HttpSelfHostServer(configuration))
    {
        server.OpenAsync().Wait();
        Console.WriteLine("按任意键关闭 Web API");
        Console.ReadLine();
    }
}

```

我们在客户端利用 `HttpClient` 进行 Web API 的调用，同样通过 Nuget 的方式添加所需的引用，对应的 Nuget Package 为“Microsoft ASPNET Web API Client Libraries (Microsoft.AspNet.WebApi.Client)”，最后编写如下的 Web API 调用程序。

```

class Program
{
    static void Main(string[] args)
    {
        Uri baseAddress = new Uri("http://127.0.0.1:3721");
        HttpClient httpClient = new HttpClient { BaseAddress = baseAddress };
        IEnumerable<Contact> contacts = httpClient.GetAsync("/api/contacts")
            .Result.Content.ReadAsAsync<IEnumerable<Contact>>().Result;
        Console.WriteLine("当前联系人列表: ");
        ListContacts(contacts);

        Contact contact = new Contact { Id = "003", Name = "王五",
            EmailAddress = "wangwu@gmail.com", PhoneNo = "789" };
        Console.WriteLine("\n 添加联系人 003: ");
        httpClient.PutAsync<Contact>("/api/contacts", contact,
            new JsonMediaTypeFormatter()).Wait();
        contacts = httpClient.GetAsync("/api/contacts")
            .Result.Content.ReadAsAsync<IEnumerable<Contact>>().Result;
        ListContacts(contacts);

        contact = new Contact { Id = "003", Name = "王五",
            EmailAddress = "zhaoliu@outlook.com", PhoneNo = "987" };
        Console.WriteLine("\n 修改联系人 003: ");
        httpClient.PostAsync<Contact>("/api/contacts", contact,
            new XmlMediaTypeFormatter()).Wait();
        contacts = httpClient.GetAsync("/api/contacts")
            .Result.Content.ReadAsAsync<IEnumerable<Contact>>().Result;
        ListContacts(contacts);

        Console.WriteLine("\n 删除联系人 003: ");
        httpClient.DeleteAsync("/api/contacts/003").Wait();
        contacts = httpClient.GetAsync("/api/contacts")
            .Result.Content.ReadAsAsync<IEnumerable<Contact>>().Result;
        ListContacts(contacts);

        Console.Read();
    }
}

```

```

    }

    private static void ListContacts(IEnumerable<Contact> contacts)
    {
        foreach (Contact contact in contacts)
        {
            Console.WriteLine("{0, -6}{1, -6}{2, -20}{3, -10}", contact.Id,
                contact.Name, contact.EmailAddress, contact.PhoneNo);
        }
    }
}

```

如上面的代码片段所示，我们先根据基地址 `http://127.0.0.1:3721` 创建了一个 `HttpClient` 对象，并调用 `GetAsync` 方法获取并显示当前联系人列表，然后分别调用方法 `PutAsync<Contact>`、`PostAsync<Contact>`和 `DeleteAsync` 对联系人作添加、修改和删除操作，为了确定操作是否成功执行，我们在每次操作执行结束之后（调用 `Task` 的 `Wait` 方法）将当前联系人列表输出来。先后启动控制台程序 `WebApi` 和 `Client`，客户端会出现如下的输出结果，可以证明所有的 `CRUD` 操作均成功执行。

当前联系人列表:

001	张三	zhangsan@gmail.com	123
002	李四	lisi@gmail.com	456

添加联系人 003:

001	张三	zhangsan@gmail.com	123
002	李四	lisi@gmail.com	456
003	王五	wangwu@gmail.com	789

修改联系人 003:

001	张三	zhangsan@gmail.com	123
002	李四	lisi@gmail.com	456
003	王五	zhaoliu@outlook.com	987

删除联系人 003:

001	张三	zhangsan@gmail.com	123
002	李四	lisi@gmail.com	456

本章小结

REST 提供一种面向资源的架构风格，提倡直接建立在 Web 上进行针对服务调用的消息交换。虽然 WCF 提供了针对 REST 的支持，但是不论从编程方式还是框架本身的设计来看，ASP.NET Web API 都是更好的选择。

ASP.NET Web API 采用了管道式设计，整个服务端管道本质上就是一个 `HttpMessageHandler` 链。处于管道首尾的分别是 `HttpServer` 和 `HttpControllerDispatcher`，它们之间可以添加任意的 `HttpMessageHandler`。作为服务端管道末端的 `HttpControllerDispatcher` 是 ASP.NET Web API 服务端框架的核心，它主导了 `HttpController` 的激活和 `Action` 的执行。

如果我们具有额外的请求/响应处理需求，可以通过自定义 `HttpMessageHandler` 来实现。

ASP.NET Web API 利用 `HttpControllerActivator` 来激活目标 `HttpController` 对象，而对 `HttpController` 类型的解析则是通过 `HttpControllerTypeResolver` 来完成的。针对当前请求的 Action 选择机制不仅仅依赖于 URL 路由，还与请求采用的 HTTP 方法有关，Action 选择由 `HttpActionSelector` 来完成。ASP.NET Web API 采用了与 ASP.NET MVC 几乎一致的 Model 元数据解析机制，解析出来的 Model 元数据最终服务于 Action 方法参数的绑定和 Model 验证。

ASP.NET Web API 的参数绑定系统由 `HttpActionBinding`、`HttpParameterBinding` 和 `ActionValueProvider` 这三个核心对象协作完成。`FormatterParameterBinding` 和 `ModelBinderParameterBinding` 提供了两种基本的参数绑定方式，前者通过反序列化从请求消息主体部分读取的内容来生成参数对象，后者则利用类似于 ASP.NET MVC 的 Model 绑定机制来提供作为参数的对象。`FormatterParameterBinding` 在进行参数绑定的时候会利用 `BodyModelValidator` 对象对请求数据实施验证。

Web API 的调用本质上就是一个 HTTP 请求发送和响应解析的过程，完全可以按照我们熟悉的网络编程来调用某个 Web API。Ajax 是在 Web 前端进行 Web API 调用的主要方式。除此之外，`HttpClient` 为托管程序提供了一个方便快捷的 Web API 调用方式。ASP.NET Web API 可以寄宿于一个 ASP.NET Web 应用中，也可以采用“自我寄宿”的方式承载于一个其他类型的应用程序。Web API 的自我寄宿通过 `HttpSelfHostServer` 来实现，其本质就是利用 `HttpBinding` 创建一个请求监听器和一个接收并处理请求的信道栈。

第 10 章 案例实践

我们通过前面的章节对整个 ASP.NET MVC 框架的请求处理机制进行了全面的讲解和深入的剖析，同时创建了一系列扩展以解决项目开发中的一些问题，比如 IoC 的集成、多规则 Model 验证和自动化异常处理等，现在我们将它们应用在一个实例中。我们把这个应用称为 Video Mall (以下简称 VM)，它是一个采用 ASP.NET MVC 构建的销售电影碟片的电子商务网站。

10.1 功能性简介

VM 是一个采用 ASP.NET MVC 创建的在线销售影碟的电子商务网站，我们采用 ADO.NET Entity Framework 作为数据访问的 ORM 框架，而数据最终存放在一个 SQL Server 数据库中。我们先通过运行过程的截图来介绍一下 VM 的基本功能。

10.1.1 商品列表的呈现

图 10-1 所示的是 VM 的主页，它按照发行时间分页显示影碟列表。每页显示的商品数量通过配置决定，默认显示第 1 页的内容。我们可以通过点击分页导航链接显示指定某页的影碟列表，而对应的 URL 格式为“~/Page{PageIndex}”（比如“~/Page2”）。

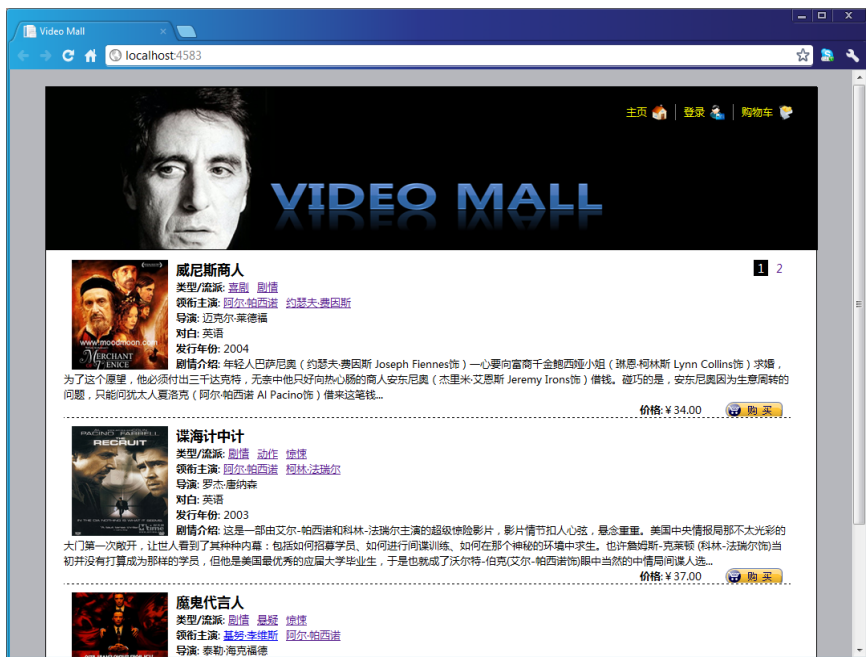


图 10-1 VM 主页

商品列表中每一项会显示某张影碟的基本信息，包括影片名称、类型、领衔主演、导演、对白语言、片长、发行年份、价格和剧情介绍摘要等。影片的类型名称和领衔主演显示为链接，意味着可以点击它们获取如图 10-2 所示的某种类型或者由某个演员主演的影片列表。和主页一样，这里的影片列表依然是分页显示的，具体的 URL 格式为“~/Genre/{Genre}/Page{PageIndex}”（比如“~/Genre/剧情/Page2”）和“~/Actor/{Actor}/Page{PageIndex}”（比如“~/Actor/阿尔·帕西诺/Page2”）。

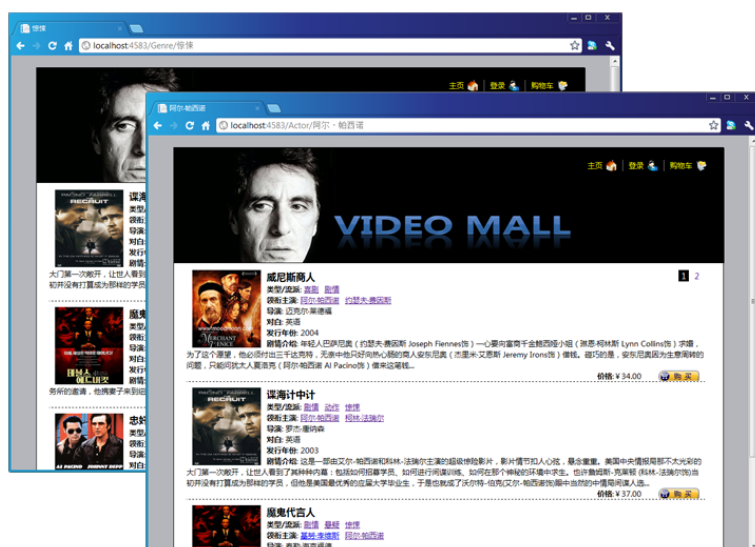


图 10-2 基于某种类型或者由某个演员主演的影片列表

对于上面三类显示影片列表的页面来说，电影海报图片是一个指向详细页面的链接。图 10-3 表示的影片详细页面的截图，可以看到它指定了影片所有相关的信息。值得一提的是，为了让 URL 具有更强的可读性和指示性，详细页面的 URL 采用的格式为“~/ {ProductName}/{ProductID}”。比如图 10-3 所在页面的地址为“~/魔鬼代言人/006”，包含在 URL 中的 ID (006) 是给程序使用的（用于获取对应的影片信息），而片名是给人看的（用户一看就知道对应的影片名称是“魔鬼代言人”）。从 SEO 的角度看，较之仅仅包含 ID 的 URL，这种包含片名的 URL 更加易于被搜索引擎收录。

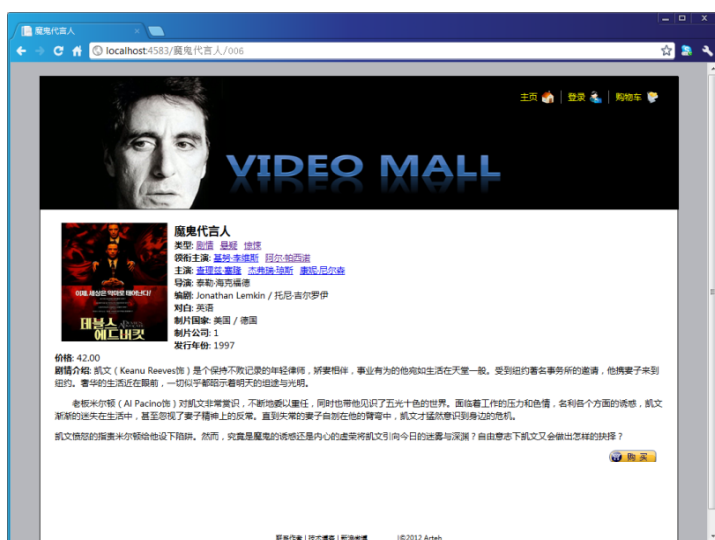


图 10-3 影片详细信息

10.1.2 定购商品

不论影片列表页面（显示所有影片列表的主页和显示针对某种类型/某个演员的影片列表）还是影片详细信息的页面，都具有一个“购买”按钮。当我们点击该按钮的时候，相应的影片将会添加到购物车中，然后如图 10-4 所示的购物车页面会立即被呈现出来。购物车页面还可以通过整个页面右上角的“购物车”链接来呈现。购物车页面的 URL 为“~/ShoppingCart”。

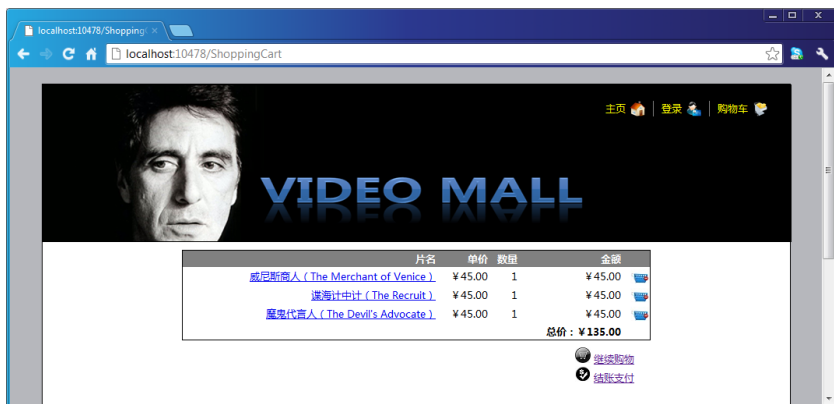


图 10-4 购物车列表

该页面列出了当前购物车中所有影片的名称、单价、数量和金额，我们可以通过影片名称对应的链接查看影片的详细信息，也可以点击购物车商品项右边的蓝色图标将该商品从购物车中移出。购物车商品列表右下方具有两个链接，点击“继续购物”会回到主页；登录用户点击“结账支付”后系统将根据购物车中的商品列表生成并提交订购订单。当订单被成功提交之后，相应的确认信息会通过如图 10-5 所示的页面显示出来，该页面对应的 URL 为“~/CheckOut”。

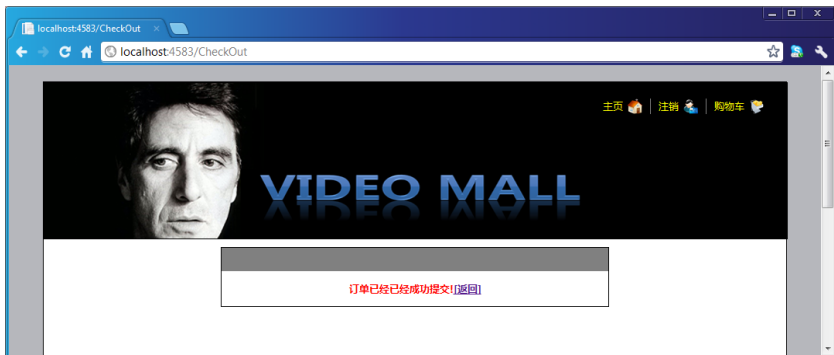


图 10-5 订单提交成功确认页面

10.1.3 登录与错误页面

如果在尚未登录的情况下点击“购物车”页面的“结账支付”按钮，会自动重定向如图 10-6 所示的登录页面。在这种情况下如果用户输入正确的用户名和密码完成登录之后，订单会被自动提交。我们也可以通过点击页面右上角的“登录”链接来进行登录，登录页面的 URL 为“~/Login”。

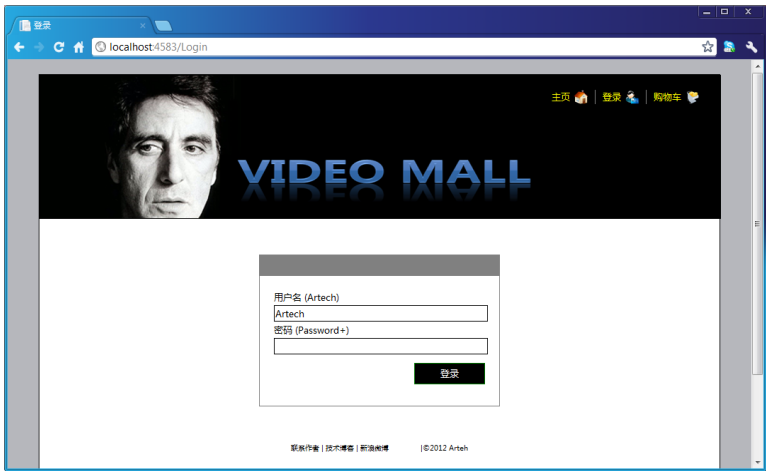


图 10-6 用户登录页面

我们将在第 7 章“Action 的执行”中实现的自动化异常处理机制应用到 VM 中，最终的错误消息主要具有两种呈现方式：一种是显示在当前页面的 ValidationSummary 中，另一种则是显示在如图 10-7 所示的单独的错误页面中。由于我们在错误页面中包含处理后异常的类型和 StackTrace，所以这样的错误页面仅限于在开发阶段使用。

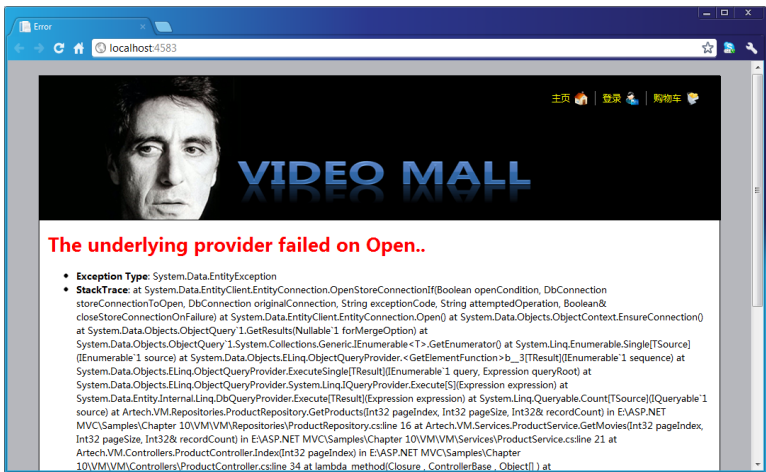


图 10-7 错误页面

10.2 设计概述

VM 提供了“商品列表呈现”、“商品订购”和“登录/注销”三个基本的功能，我们希望 VM 对于真正的项目开发具有参考价值，所以层次化和模块化的设计体现在这个简单的 ASP.NET MVC 应用中。

10.2.1 Controller-Service-Repository

整个 VM 基本的层次结构可以通过图 10-8 来体现。VM 采用一个 SQL Server 数据库作为数据存储，并且利用了 ADO.NET Entity Framework 作为进行数据访问的 ORM 框架。具体来说，我们采用的是 POCO（Plain Old CLR Objects）的开发方式，DbContext 帮助我们实现与数据库的交互。我们采用了 Repository 模式将单纯的数据访问操作封装在 Repository 中，它可以看成是针对某个 Entity Data Model（通过一个.edmx 文件进行定义）的 DbContext 的封装。

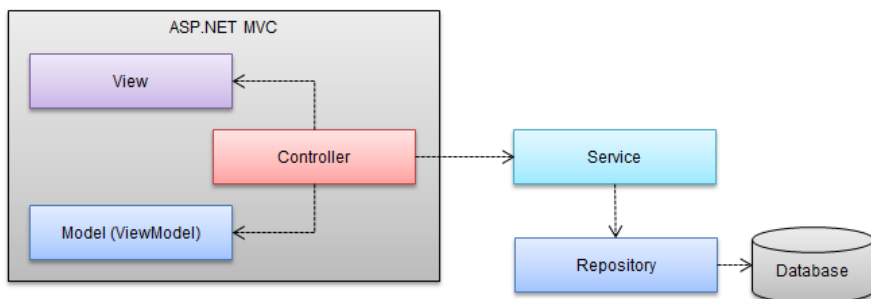


图 10-8 VM 基本的层次结构

业务逻辑实现在 Service 中并以服务的形式暴露出来，Service 调用 Repository 完成针对数据的存取操作。在实际项目开发中，我们可以定义独立的层次来定义业务逻辑，然后封装成服务对外发布，由于 VM 中并没有复杂的业务逻辑，我们直接将服务和业务逻辑合二为一，服务的消费者是 ASP.NET MVC 的 Controller。

基于接口的调用

对于 Controller-Service-Repository 层次结构来说，上层对于下层的调用都是基于接口来完成的。具体来说，每一个具体的 Service 和 Repository 均具有相应的接口，Controller 通过 Service 的接口来调用 Service，Service 通过 Repository 的接口来调用 Repository，具体的调用关系如图 10-9 所示。

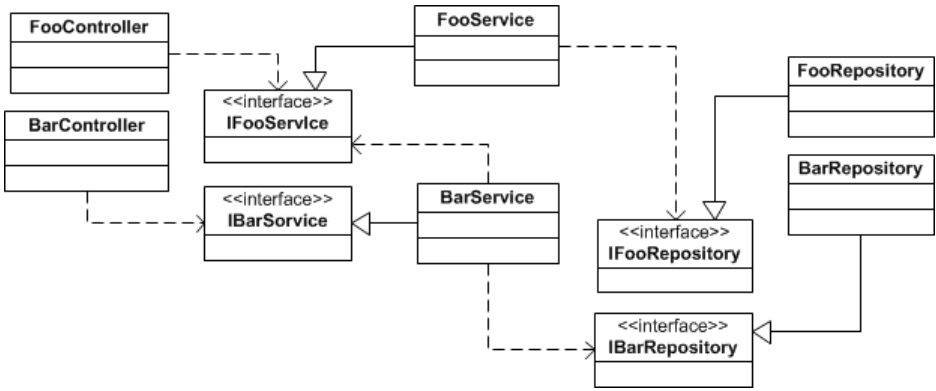


图 10-9 基于接口的层次调用

接口是一种功能消费的契约，两个层次之间通过预定义的接口进行交互能够最大限度地降低两个层次之间的耦合度。除此之外，基于接口的层次调用带来的另一个好处就是使测试变得容易了。由于被测试组件对另一个依赖组件的调用是通过接口来实现的，我们可以很容易地 Mock 一个实现了该接口的对象来对被依赖的组件进行独立测试。也就是说我们可以通过对 Service 的 Mock 来独立测试定义在 Controller 的 UI 处理逻辑（无需考虑真正的业务处理逻辑），通过对 Repository 的 Mock 来独立测试定义在 Service 中的业务逻辑（无需考虑具体的数据存储）。

除了基于接口的上下层之间的调用，图 10-9 还揭示了另一个细节：处于同一层次之间的 Service 可以通过接口进行相互调用（BarService 可以通过 IFooService 调用 FooService）。这也很好理解，某个模块的业务逻辑可能涉及到对另一个模块的调用。

但是我们不赞成一个 Service 以接口的形式调用另一个 Service 的 Repository，在这里我们将 Service 视为 Repository 的拥有者，Repository 辅助 Service 完成整个业务流程的实现。一个业务逻辑单元对外的接口应该只有一个，那就是 Service 实现的接口。

Repository

我们来简单地介绍一下 Repository 在 VM 中的设计。将 Repository 视为操作某种实体类型数据的组件，为此我们定义了一个泛型的 IRepository<TEntity> 类型，泛型参数表示实体类型，通过如下所示的代码在 IRepository<TEntity> 中定义了一些基于 CRUD 操作的方法。

```
public interface IRepository<TEntity> : IDisposable where TEntity : class
{
    IEnumerable<TEntity> Get();
    IEnumerable<TEntity> Get(Expression<Func<TEntity, bool>> filter);
    IEnumerable<TEntity> Get<TOrderKey>(Expression<Func<T, bool>> filter,
        int pageIndex, int pageSize,
        Expression<Func<TEntity, TOrderKey>> sortExpression, bool isAsc = true);
}
```



```

    int Count(Expression<Func<TEntity, bool>>filter);
    void Add(TEntity instance);
    void Update(TEntity instance);
    void Delete(TEntity instance);
}

```

`IRepository<TEntity>` 的 `Add`、`Update` 和 `Delete` 实现基本的数据添加、更新和删除功能。`Count` 方法用于返回筛选数据的数量，筛选条件通过类型为 `Expression<Func<TEntity, bool>>` 的表达式参数 `filter` 表示。三个 `Get` 方法用于获取相应的数据，其中第一个重载用于获取所有的数据，而其余两个则可以指定筛选条件。最后一个 `Get` 方法在第二个的基础上增加了分页获取的功能，我们可以通过参数指定每页的大小（`pageSize`）、获取页码（`pageIndex`）、排序表达式（`sortExpression`）和排序类型（`isAsc`：升序还是降序）。

`Repository` 利用 ADO.NET Entity Framework 实现最终的数据存取，所以我们需要通过连接数据库生成一个实体数据模型（最终体现为一个 `.edmx` 文件）。VM 只涉及三个业务表，分别是用于存储商品数据的 `Product`，和存储订单数据的 `Order`（主表）和 `OrderLine`（子表）。在根据三个数据表为整个 VM 应用创建实体数据模型之后，我们利用自动的代码生成机制生成三个单纯的实体类型 `Product`、`Order` 和 `OrderLine`，关于数据表的结构、实体数据模型和实体类型的生成，我们会在下一节进行单独介绍。

我们针对实体数据模型定义一个继承自 `System.Data.Entity.DbContext` 的 `VmDbContext`（实际上这个类型和实体类型一样可以通过相应的代码生成机制自动生成，但是我们涉及到的实体类型很少，所以选择自行定义）。如下面的代码片段所示，我们在构造函数中指定了对应的数据库连接字符串，并为基于对应数据表的实体类型定义了 `DbSet<TEntity>` 属性。

```

public class VmDbContext: DbContext
{
    public VmDbContext()
        : base("name=VM")
    {
    }

    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderLine> OrderLines { get; set; }
    public DbSet<Product> Products { get; set; }
}

```

我们为 VM 中所有的 `Repository` 定义了一个抽象基类 `VmRepository<TEntity>`。如下面的代码片段所示，`VmRepository<TEntity>` 实现了 `IRepository<TEntity>` 接口，它的核心是通过只读属性 `DbContext` 表示的 `VmDbContext` 对象，而 `DbSet` 属性是通过 `VmDbContext` 的 `Set` 方法设置的针对当前实体类型的 `DbSet` 对象。如果读者对 ADO.NET Entity Framework 中的 `DbContext` 具有基本的了解，相信对在 `IRepository<TEntity>` 接口定义的成员在 `VmRepository<TEntity>` 中的实现不会感到陌生。最后值得一提的是，我们创建的 `VmDbContext` 会在实现的 `Dispose` 方法中被释放。

```

public abstract class VmRepository : IRepository
where TEntity : class
{
    public VmDbContext DbContext { get; private set; }
    public DbSet DbSet { get; private set; }
    public VmRepository(VmDbContext context)
    {
        Guard.ArgumentNotNull(context, "context");
        this.DbContext = context;
        this.DbSet = this.DbContext.Set();
    }

    public IEnumerable Get()
    {
        return this.DbSet.AsQueryable();
    }

    public IEnumerable Get(Expression> filter)
    {
        return this.DbSet.Where(filter).AsQueryable();
    }

    public IEnumerable Get<TKey>(Expression

```

```

public void Update(TEntity instance)
{
    Guard.ArgumentNotNull(instance, "instance");
    this.DbSet.Attach(instance);
    this.DbContext.Entry(instance).State = EntityState.Modified;
    this.DbContext.SaveChanges();
}

public void Delete(TEntity instance)
{
    Guard.ArgumentNotNull(instance, "instance");
    this.DbSet.Attach(instance);
    this.DbContext.Entry(instance).State = EntityState.Deleted;
    this.DbContext.SaveChanges();
}

public void Dispose()
{
    this.DbContext.Dispose();
}
}

```

10.2.2 IoC 的应用

VM 采用了 Controller-Service-Repository 这样的层次结构，相邻层次之间采用接口进行交互，那么对于采用基于接口的方式来提供被依赖的对象呢？IoC 无疑是一个很好的解决方案。

在第 3 章“Controller 的激活”中，我们提供了三种与 IoC 框架的集成方式（自定义 ControllerFactory、ControllerActivator 和 DependencyResolver）实现了针对 Controller 激活过程的依赖注入。我们将之前创建的基于 Unity 的自定义 UnityControllerFactory 直接应用到 VM 中。如果预先在 Unity 容器中注册了接口与实现类型之间的映射关系，那么当 Controller 在被激活过程中以各种方式定义的依赖注入（构造器注入、属性注册和方法注入）都能自动地完成。

如下面的代码片段所示，FooController 定义了一个类型为 IFooService 的只读属性 Service，该属性在构造函数中被初始化（构造器注入）。实现了 IFooService 接口的 FooService 具有一个依赖属性（应用了 DependencyAttribute 特性）Repository（体现了属性注入），其类型为 IFooRepository。如果我们在 UnityControllerFactory 使用的 Unity 容器中注册了 IFooService 与 FooService，以及 IFooRepository 和 FooRepository 之间的类型映射，那么 UnityControllerFactory 在激活 FooController 的时候会自动初始化它的 Service 属性。

```

public class FooController : Controller
{
    //其他成员
    public IFooService Service{ get; private set; }
    public ProductController(IFooService service)
    {
        this.Service= service;
    }
}

```

```

}

public class FooService: IFooService
{
    //其他成员
    [Dependency]
    public IFooRepository Repository{ get; set; }
}

```

对象的释放

ASP.NET MVC 默认采用了基于单一请求 Controller 激活和释放机制，换句话说，对于每一个 HTTP 请求，ASP.NET MVC 总是会创建一个全新的 Controller 对象，当 Controller 完成使命之后总是会自动被释放。Controller 的激活和释放都是通过 ControllerFactory 来完成的。

另一方面，Repository 基本上可以看成是对 DbContext 的封装。Repository 实现了 IDisposable 接口将 DbContext 的释放实现在 Dispose 方法中。我们希望的是当 Controller 被释放的时候，对应的 Repository 能够自动释放。但是 Controller 是通过 Service 间接地引用 Repository 的，所以 Controller 对 Repository 的释放需要间接地通过 Service 来完成。

为此我们为 Service 定义了一个抽象基类 ServiceBase。如下面的代码片段所示，ServiceBase 具有一个元素类型为 IDisposable 的列表的属性 DisposableObjects，它表示被它引用并且由它负责释放的对象列表。方法 AddDisposableObject 负责将指定的对象放入该列表，ServiceBase 实现了 IDisposable 接口，并在 Dispose 方法中对 DisposableObjects 列表中的所有对象进行释放。

```

public abstract class ServiceBase : IDisposable
{
    public IList<IDisposable> DisposableObjects { get; private set; }

    public ServiceBase()
    {
        this.DisposableObjects = new List<IDisposable>();
    }

    protected void AddDisposableObject(object obj)
    {
        IDisposable disposable = obj as IDisposable;
        if (null != disposable)
        {
            this.DisposableObjects.Add(disposable);
        }
    }

    public void Dispose()
    {
        foreach (IDisposable obj in this.DisposableObjects)
        {
            if (null != obj)

```

```

        {
            obj.Dispose();
        }
    }
}

```

我们在通过继承 `ServiceBase` 定义具体的 `Service` 的时候，如果 `Service` 引用着一些可被释放的对象，可以通过调用 `AddDisposableObject` 方法将其放到 `DisposableObjects` 列表中，以确保自身在被释放的时候这些对象能够得以正常释放。这些被 `Service` 引用的对象自然就包括 `Repository` 对象，我们可以通过如下的方式来定义包含 `Repository` 的 `Service` 类型。

```

public class FooService: ServiceBase, IFooService
{
    //其他成员
    public IFooRepository Repository{ get; private set; }
    public FooService(IFooRepository repository)
    {
        this.Repository = repository;
        base.AddDisposableObject(repository);
    }
}

```

`Repository` 通过 `Service` 释放，而对后者的释放在 `Controller` 的释放时自动完成，所以我们将相同的设计引入到 `Controller` 上，所以我们为整个 VM 应用的所有 `Controller` 定义一个基类 `VmController`。如下面的代码片段所示，`VmController` 的基类是为了实现自动化异常处理而定义的 `ExtendedController`，`VmController` 同样定义了 `DisposableObjects` 属性和 `AddDisposableObject` 方法，唯一不同的是，具体的对象释放实现在重写的受保护的 `Dispose` 方法（具有一个布尔类型参数）中。

```

public class VmController: ExtendedController
{
    public IList<IDisposable> DisposableObjects { get; private set; }
    public VmController()
    {
        this.DisposableObjects = new List<IDisposable>();
    }

    protected void AddDisposableObject(object obj)
    {
        IDisposable disposable = obj as IDisposable;
        if (null != disposable)
        {
            this.DisposableObjects.Add(disposable);
        }
    }

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            foreach (IDisposable obj in this.DisposableObjects)
            {
                if (null != obj)
                {

```

```

        obj.Dispose();
    }
}
base.Dispose(disposing);
}
}

```

10.2.3 AOP 的应用

AOP 的诞生并不算晚，早在 1990 年来自 Palo Alto Research Lab (PARC: 施乐帕克研究中心) 的研究人员就面对面向对象思想的局限性进行了分析。他们研究出了一种新的编程思想，借助这一思想可以通过减少代码重复模块从而帮助开发人员提高工作效率，这个编程思想就是 AOP。随着研究的逐渐深入，AOP 也逐渐发展成一套完整的程序设计思想，各种应用 AOP 的技术也应运而生。

AOP (AspectOriented Programming, 面向方面编程)，可以说是 OOP (Object-Oriented Programing, 面向对象编程) 的补充和完善。OOP 引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP 则显得无能为力。OOP 允许我们定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能，日志代码往往水平地散布在所有对象层次中，而与核心功能毫无关系，对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切 (Crosscutting) 代码，在 OOP 设计中它会导致大量代码的重复，而不利于各个模块的重用。

AOP 技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为“Aspect” (方面)。所谓“方面”，简单地说，就是将那些与业务无关，却被业务模块所共同使用的逻辑封装起来，便于减少系统的重复代码，降低模块间的耦合度并有利于未来的可操作性和可维护性。

我们在 VM 中采用了 Unity 这个 IoC 框架，而它通过一个叫做 Interception 的扩展提供了 AOP 的实现。考虑到很多读者对 Unity 及其 Interception 扩展不太熟悉，我们通过一个简单的实例来演示 AOP 的作用和基于 Unity 的实现。

实例演示：通过 AOP 的方式实现针对方法返回值的缓存 (S1001)

对于一些用于获取数据的方法，如果数据本身发生改变的频率较低，而且获取数据本身是一个相对耗时的操作，我们倾向于对方法返回值进行缓存。在接下来这个例子中，我们利用 Unity 的 Interception 扩展以 AOP 的方式实现针对某个方法的返回值的自动缓存。

AOP 的本质在于拦截针对目标方法的调用以执行一些额外的操作，对于 Unity 的 Interception 扩展来说，我们需要将这些动态执行的操作定义在一个名为 CallHandler 的对象

中。针对方法返回值的缓存就定义在一个具有如下定义的 `CachingCallHandler` 中。

```
public class CachingCallHandler : ICallHandler
{
    public TimeSpan ExpirationTime { get; private set; }
    internal static TimeSpan DefaultExpirationTime {get; private set;}
    internal static Func<MethodBase, object[], string> CacheKeyGenerator
        { get; private set; }

    static CachingCallHandler()
    {
        DefaultExpirationTime = new TimeSpan(0, 5, 0);
        Guid prefix = Guid.NewGuid();
        //缓存项 Key 的格式: {GUID}: {方法返回声明类型名称}{方法名称}: {输入参数 HashCode}
        CacheKeyGenerator = (method, inputs)=>
        {
            StringBuilder sb = new StringBuilder();
            sb.AppendFormat("{0}:", prefix);
            sb.AppendFormat("{0}:", method.DeclaringType.FullName);
            sb.AppendFormat("{0}:", method.Name);
            if (null != inputs)
            {
                Array.ForEach(inputs, input=>
                {
                    string hashCode = (
                        null== input)?"":input.GetHashCode().ToString();
                    sb.AppendFormat("{0}:", hashCode);
                });
            }
            return sb.ToString().TrimEnd(':');
        };
    }

    public CachingCallHandler(TimeSpan? expirationTime = null)
    {
        this.ExpirationTime = expirationTime.HasValue ?
            expirationTime.Value : DefaultExpirationTime;
    }

    public IMethodReturn Invoke(IMethodInvocation input,
        GetNextHandlerDelegate getNext)
    {
        MethodInfo targetMethod = (MethodInfo)input.MethodBase;
        if (targetMethod.ReturnType == typeof(void))
        {
            return getNext()(input, getNext);
        }
        object[] inputs = new object[input.Inputs.Count];
        input.Inputs.CopyTo(inputs, 0);
        string cacheKey = CacheKeyGenerator(targetMethod, inputs);
        object[] cachedResult = HttpRuntime.Cache.Get(cacheKey) as object[];
        if (null == cachedResult)
        {
            IMethodReturn realReturn = getNext()(input, getNext);
            if (null == realReturn.Exception)
            {
                HttpRuntime.Cache.Insert(cacheKey,
                    new object[] { realReturn.ReturnValue }, null,
```

```

        DateTime.Now.Add(this.ExpirationTime),
        Cache.NoSlidingExpiration);
    }
    return realReturn;
}
return input.CreateMethodReturn(cachedResult[0],
    new object[] { input.Arguments });
}

public int Order { get; set; }
}

```

可能读者之前不曾了解过 Unity，对 `CallHandler` 的定义更是不太清楚，不过应该能够大致看明白实现在 `Invoke` 方法中针对方法返回值的缓存机制：当目标方法的调用被拦截并调用 `CachingCallHandler` 的时候，我们根据方法名称、方法返回类型和参数列表生成一个 `Key`，试图根据这个 `Key` 从缓存中获取表示返回值的对象。如果缓存的返回值存在，则直接将其作为本次方法调用的返回值；如果不存在，则执行目标方法（我们可以简单地将 `getNext()(input, getNext)` 视为针对目标方法的调用），并将返回的结果进行缓存。

从上面的代码不难看出，我们在对方法返回值进行缓存的时候采用了基于“绝对时间”的缓存过期策略。`CachingCallHandler` 具有一个 `Nullable<TimeSpan>` 类型的只读属性 `ExpirationTime` 表示过期时间，默认情况下是 5 分钟。

将 `CachingCallHandler` 应用到目标方法具有两种方式，一种是在目标类型或者方法上应用 `CallHandler` 对应的特性，另一种则是通过配置的方式。在这里我们采用前一种方式，所以我们还需要为 `CachingCallHandler` 定义如下一个 `CachingCallHandlerAttribute` 特性。该特性继承自 `HandlerAttribute`，并在实现的 `CreateHandler` 方法中实现了针对 `CachingCallHandler` 的创建。`CachingCallHandlerAttribute` 的构造函数具有一个字符串类型的参数用于指定缓存过期时间。

```

[AttributeUsage( AttributeTargets.Class| AttributeTargets.Interface|
    AttributeTargets.Method)]
public class CachingCallHandlerAttribute : HandlerAttribute
{
    public TimeSpan? ExpirationTime { get; private set; }
    public CachingCallHandlerAttribute(string expirationTime = "")
    {
        if (!string.IsNullOrEmpty(expirationTime))
        {
            TimeSpan expirationTimeSpan;
            if (!TimeSpan.TryParse(expirationTime, out expirationTimeSpan))
            {
                throw new ArgumentException("输入的过期时间 (TimeSpan) 不合法",
                    "expirationTime");
            }
            this.ExpirationTime = expirationTimeSpan;
        }
    }

    public override ICallHandler CreateHandler(IUnityContainer container)
    {
        return new CachingCallHandler(this.ExpirationTime)
    }
}

```



```

        { Order = this.Order };
    }
}

```

为了演示 `CachingCallHandler` 实现的自动缓存效果，我们在一个 ASP.NET MVC 应用添中加了针对 Unity 及其 `Interception` 扩展的程序集引用，然后定义了如下一个 `ITimeProvider` 接口，其唯一的 `GetCurrentTime` 方法返回指定 `DateTimeKind` 的当前时间，而 `DefaultTimeProvider` 实现了该接口。`DefaultTimeProvider` 的 `GetCurrentTime` 方法上应用了 `CachingCallHandlerAttribute` 特性并将缓存过期时间设置为 3 秒。

```

public interface ITimeProvider
{
    DateTime GetCurrentTime(DateTimeKind dateTimeKind);
}

public class DefaultTimeProvider : ITimeProvider
{
    [CachingCallHandler("00:00:03")]
    public DateTime GetCurrentTime(DateTimeKind dateTimeKind)
    {
        switch (dateTimeKind)
        {
            case DateTimeKind.Local: return DateTime.Now.ToLocalTime();
            case DateTimeKind.Utc: return DateTime.Now.ToUniversalTime();
            default: return DateTime.Now;
        }
    }
}

```

然后将如下所示的针对 Unity 的配置定义在 `Web.config` 中。在该配置中，我们定义了一个应用了 `Interception` 扩展（`<extension type="Interception"/>`）的 Unity 容器。该容器注册了 `ITimeProvider` 和 `DefaultTimeProvider` 之间的类型映射，并且将拦截器类型设置为 `InterfaceInterceptor`。

拦截器的类型决定了 AOP 的实现机制，Unity 的 `Interception` 扩展定义了三种拦截器：`InterfaceInterceptor`（动态生成实现了指定接口的类型，注入的操作定义在实现的方法中）、`TransparentProxyInterceptor`（采用 `TransparentProxy` 的拦截机制）和 `VirtualMethodInterceptor`（动态目标类型的子类，通过重写虚方法的方式实现对注入的操作的执行）。配置元素 `<policyInjection/>` 表示采用 `Policy Injection`（即动态执行 `CallHandler` 的方式）进行注入操作的执行。

```

<configuration>
  <configSections>
    <section name="unity"
      type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
        Microsoft.Practices.Unity.Configuration"/>
  </configSections>
  <unity>
    <sectionExtension
      type="Microsoft.Practices.Unity.InterceptionExtension.Configuration
        .InterceptionConfigurationExtension,
        Microsoft.Practices.Unity.Interception.Configuration" />
  </unity>
</configuration>

```

```

<containers>
  <container>
    <extension type="Interception" />
    <register type="MvcApp.ITimeProvider, MvcApp"
      mapTo="MvcApp.DefaultTimeProvider, MvcApp">
      <interceptor type="InterfaceInterceptor" />
      <policyInjection />
    </register>
  </container>
</containers>
</unity>
</configuration>

```

接下来我们定义了如下一个 HomeController。类型为 ITimeProvider 的 TimeProvider 是一个依赖属性，在默认的 Action 方法 Index 中我们利用该属性获取当前时间（UTC 或者 Local）并将得到的时间呈现出来。时间的获取和呈现在一个 For 循环中进行，每次迭代的时间间隔为 1 秒。

```

public class HomeController : Controller
{
    [Dependency]
    public ITimeProvider TimeProvider { get; set; }

    public void Index()
    {
        for (int i = 0; i < 3; i++)
        {
            Response.Write(string.Format("{0}: {1: hh:mm:ss}<br/>", "Utc",
                this.TimeProvider.GetCurrentTime(DateTimeKind.Utc)));
            Thread.Sleep(1000);

            Response.Write(string.Format("{0}: {1: hh:mm:ss}<br/><br/>",
                "Local", this.TimeProvider.GetCurrentTime(DateTimeKind.Local)));
            Thread.Sleep(1000);
        }
    }
}

```

由于通过 CachingCallHandler 实现的缓存是通过 Unity 的 Interception 扩展实现的，所以需要采用 Unity 的方式来激活 Controller，在这里我们在 Global.asax 中通过如下的代码将在第 3 章“Controller 的激活”中定义的 UnityControllerFactory 进行了注册¹。

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        ControllerBuilder.Current.SetControllerFactory(
            new UnityControllerFactory());
    }
}

```

现在直接运行我们的程序会在浏览器中呈现出如图 10-10 所示的输出结果，针对方法返回值的缓存得到了很明显体现。输出结果还体现 CachingCallHandler 是根据方法的输入参数

¹ 第 3 章“Controller 的激活”中定义 UnityControllerFactory 采用的是针对编程的类型映射方式，这里对它作了改进，使我们可以通过配置的方式进行类型注册，这个改进后的 UnityControllerFactory 使用在 VM 之中。

进行缓存的(参数为 `DateTimeKind.Local` 的方法调用不会使用针对 `DateTimeKind.Utc` 参数值的缓存结果), 并且缓存过期时间为 3 秒(第 1 次迭代缓存值被第 2 次迭代使用, 但是在进行第 3 次迭代时就已经过期了)。

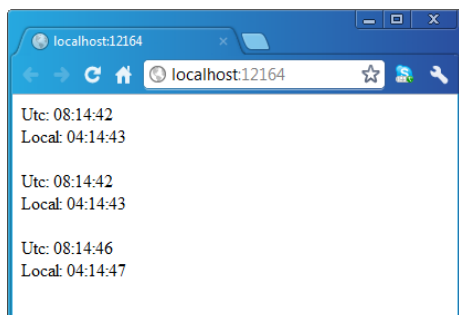


图 10-10 CachingCallHandler 对方法返回值(时间)的缓存

TransactionCallHandler

我们在 VM 中定义了如下一个 `TransactionCallHandler` 和对应的 `TransactionCallHandlerAttribute` 特性以实现声明式的事务控制。根据定义我们不难看出, 当我们将 `TransactionCallHandlerAttribute` 特性应用到某个方法上时, 如果被 `UnityContainer` 创建的对象支持拦截机制, 针对该方法将会在 `TransactionScope` 中执行。

```
public class TransactionCallHandler: ICallHandler
{
    public IMethodReturn Invoke(IMethodInvocation input,
        GetNextHandlerDelegate getNext)
    {
        try
        {
            using (TransactionScope transactionScope = new TransactionScope())
            {
                IMethodReturn methodReturn = getNext()(input, getNext);
                transactionScope.Complete();
                return methodReturn;
            }
        }
        catch (Exception ex)
        {
            return input.CreateExceptionMethodReturn(ex);
        }
    }

    public int Order { get; set; }
}

[AttributeUsage(AttributeTargets.Method)]
public class TransactionCallHandlerAttribute: HandlerAttribute
{
    public override ICallHandler CreateHandler(IUnityContainer container)
    {

```

```

        return new TransactionCallHandler { Order = this.Order };
    }
}

```

10.2.4 异常处理

在第7章“Action的执行”中我们通过重写 `OnException` 方法的自定义 `Controller` 类型实现了“自动化”异常处理。具体来说，在重写的 `OnException` 方法中我们借助 `EntLib` 的 `Exception Handling Application Block` 根据指定的策略自动处理抛出的异常，而经过处理后的错误消息则采用如下的方式呈现出来。

- 如果当前是 Ajax 请求，则根据处理后的异常生成一个 `JsonResult` 对响应予以请求。
- 如果当前 Action 方法通过 `HandleErrorActionAttribute` 特性设置了 `Handle-Error-Action`，那么对应的 Action 方法会被执行，而返回的 `ActionResult` 将用于对请求的响应。
- 在其他情况下，默认的错误 View 会被呈现出来作为对请求的响应。

如下面的代码片段所示，作为所有 `Controller` 基类的 `VmController` 正是实现了自动化异常处理的 `ExtendedController` 的子类，上面应用 `ExceptionPolicyAttribute` 特性设置了默认采用异常处理策略（`defaultPolicy`）。

```

[ExceptionPolicy("defaultPolicy")]
public class VmController : ExtendedController
{
    //省略成员
}

```

如下所示的是 VM 采用的异常处理策略的配置，我们仅仅针对性地处理三种异常类型，即 `InvalidUserNameException`、`InvalidPasswordException` 和 `OutOfStockException`。前两种类型的异常在登录时输入错误的用户名或者密码时抛出，而当某个商品库存量少于订购数量时，`OutOfStockException` 异常会在提交订单的时候抛出。至于具体采用的异常处理方式，我们仅仅是通过 `ErrorMessageHandler` 设置了一个友好的错误消息而已，真正在项目开发过程中需要根据实际的需求设置处理的异常类型和异常处理方式。

```

<configuration>
  <configSections>
    <section name="exceptionHandling"
      type="Microsoft.Practices.EnterpriseLibrary.ExceptionHandling
        .Configuration.ExceptionHandlingSettings,
        Microsoft.Practices.EnterpriseLibrary.ExceptionHandling" />
  </configSections>
  <exceptionHandling>
    <exceptionPolicies>
      <add name="defaultPolicy">
        <exceptionTypes>
          <add name="InvalidUserNameException"
            type="Artech.VM.Models.InvalidUserNameException, Artech.VM"
            postHandlingAction="ThrowNewException" />

```

```

        <exceptionHandlers>
            <add name="ErrorMessageHandler"
                type="Artech.Mvc.Extensions.ErrorMessageHandler,
                    Artech.Mvc.Extensions"
                errorMessage="用户名不存在" />
        </exceptionHandlers>
    </add>
    <add name="InvalidPasswordException"
        type="Artech.VM.Models.InvalidPasswordException, Artech.VM"
        postHandlingAction="ThrowNewException" >
        <exceptionHandlers>
            <add name="ErrorMessageHandler"
                type="Artech.Mvc.Extensions.ErrorMessageHandler,
                    Artech.Mvc.Extensions"
                errorMessage="密码与用户名不匹配" />
        </exceptionHandlers>
    </add>
    <add name="OutOfStockException"
        type="Artech.VM.Models.OutOfStockException, Artech.VM"
        postHandlingAction="ThrowNewException" >
        <exceptionHandlers>
            <add name="ErrorMessageHandler"
                type="Artech.Mvc.Extensions.ErrorMessageHandler,
                    Artech.Mvc.Extensions"
                errorMessage="库存不足! " />
        </exceptionHandlers>
    </add>
</exceptionTypes>
</add>
</exceptionPolicies>
</exceptionHandling>
</configuration>

```

10.3 编程实现

前面我们从功能和设计方面对 VM 进行了概括性的介绍,现在来详细介绍具体的编程实现,包括数据表的设计和 Repository、Service、Controller 和 View 的定义。

10.3.1 数据表的创建

由于 VM 仅提供两个基本的功能(商品列表的呈现和订单的提交),所以数据库中只包含三个基本的数据表,即用于存储商品信息的 Product 表和存储订单的 Order 表(主表)和 OrderLine 表(子表),这三个数据表的结构可以通过如下所示的创建它们的 SQL 脚本来体现。

```

--商品表 (Product)
CREATE TABLE [Product] (
    [ProductId]          [VARCHAR] (50)          NOT NULL, --ID
    [Name]               [NVARCHAR] (256)       NOT NULL, --片名
    [Genre]              [NVARCHAR] (50)        NOT NULL, --影片类型
    [Starring]           [NVARCHAR] (50)        NOT NULL, --领衔主演

```

```

[SupportingActors]      [VARCHAR] (256)          NOT NULL, --主演
[Director]              [NVARCHAR] (50)         NOT NULL, --导演
[ScriptWriter]          [NVARCHAR] (50)         NOT NULL, --编剧
[ProductionCountry]     [NVARCHAR] (50)         NOT NULL, --制片国家
[ProductionCompany]     [NVARCHAR] (50)         NOT NULL, --制片公司
[ReleaseYear]           [INT]                   NOT NULL, --发行年份
[Language]              [NVARCHAR] (50)         NOT NULL, --对白语言
[RunTime]               [INT]                   NOT NULL, --片长 (分钟)
[Price]                 [decimal] (18, 0)       NOT NULL, --单价
[Poster]                [VARCHAR] (50)          NOT NULL, --电影海报图片名称
[Stock]                 [INT]                   NOT NULL, --库存
[Story]                 [NVARCHAR] (max)        NOT NULL, --剧情介绍 (完整)
[StoryAbstract]         [NVARCHAR] (1000)       NOT NULL, --剧情介绍 (摘要)
CONSTRAINT [C_Product_PK]
    PRIMARY KEY CLUSTERED ( [ProductId] ASC )
    ON [PRIMARY]) ON [PRIMARY]

--订单表 (Order) :
CREATE TABLE [Order] (
    [OrderId]             [VARCHAR] (50)          NOT NULL, --订单 ID
    [UserName]            [VARCHAR] (50)          NOT NULL, --客户账号
    [OrderTime]           [DATETIME]              NOT NULL, --订单提交时间
    CONSTRAINT [C_Order_PK]
        PRIMARY KEY CLUSTERED ( [OrderId] ASC )
        ON [PRIMARY]) ON [PRIMARY]

--订单明细表 (OrderLine)
CREATE TABLE [OrderLine] (
    [OrderId]             [VARCHAR] (50)          NOT NULL, --订单 ID
    [ProductId]           [VARCHAR] (50)          NOT NULL, --商品 ID
    [Quantity]            [INT]                   NOT NULL, --订购数量
    CONSTRAINT [C_OrderLine_PK]
        PRIMARY KEY CLUSTERED ( [ProductId] ASC, [OrderId] ASC )
        ON [PRIMARY]) ON [PRIMARY]

ALTER TABLE [OrderLine] WITH CHECK ADD CONSTRAINT [FK_OrderLine_Order]
    FOREIGN KEY ([OrderId])
    REFERENCES [Order] ([OrderId])

ALTER TABLE [dbo].[OrderLine] WITH CHECK ADD CONSTRAINT [FK_OrderLine_Product]
    FOREIGN KEY ([ProductId])
    REFERENCES [dbo].[Product] ([ProductId])

```

我们直接利用 ASP.NET 的 Membership 模块来实现的用户登录功能。由于使用的 MembershipProvider 类型为 SqlMembershipProvider，所以我们以命令行的方式执行 aspnet_regsql.exe 命令来生成相关的数据表和存储过程，然后执行如下一段 SQL 脚本在用于存储应用列表的 aspnet_Applications 表中插入一笔表示当前应用 (VM) 的记录²。

2 我们提供的 VM 源代码中包含了整个数据库文件 (“\VM\App_Data\ASPNETDB.MDF”)，它不仅包含了我们上面创建的三个数据表和响应的数据，还包含通过 aspnet_regsql.exe 命令创建的所有数据对象。除此之外，创建三个业务表和添加商品数据的 SQL 脚本被包含在相同的目录下 (“\VM\App_Data\script.sql”)。

```

INSERT INTO [dbo].[aspnet_Applications]
    ([ApplicationName]
    , [LoweredApplicationName]
    , [ApplicationId]
    , [Description])
VALUES
    ('VM'
    , 'vm'
    , '85562079-171F-4E80-8E8E-929221720F78'
    , 'Video Mall')

```

10.3.2 Repository

在第一节我们已经对 Repository 作了简单的介绍，并且给出了基类 VmRepository <TEntity>及其实现的接口 IRepository<TEntity>的定义。现在需要做的就是利用 Visual Studio 提供的向导根据三张数据表来创建相应的实体数据模型，用于定义该模型的 VM.edmx 文件在 Visual Studio 设计器体现出来的实体结构和相互关系如图 10-11 所示。

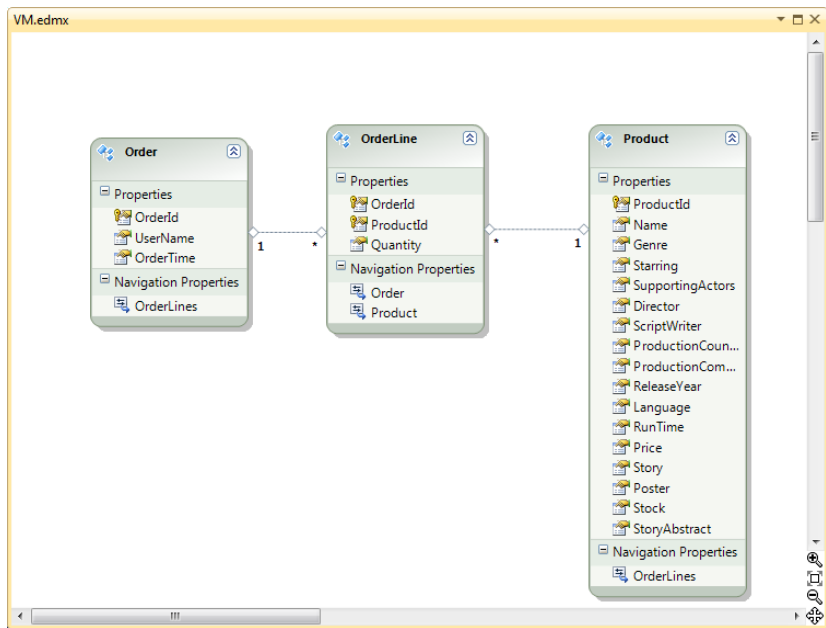


图 10-11 实体数据模型

实体类的生成和 VmDbContext 的创建

我们采用的是 POCO 的编程模式，可以直接利用 Visual Studio 提供的代码生成机制来根据上面的实体数据模型来创建相应的实体类。我们在如图 10-11 所示的实体数据模型设计器上点击右键，从弹出的上下文菜单中选择“添加代码生成选项... (Add code generation item...)”，如图 10-12 所示的“添加新项目 (Add New Item)”对话框会显示出来，然后选择

“在线模板 (Online Templates)”，在右侧选择“EF 4.x POCO Entity Generator for C#”并点击“添加 (Add)”按钮，一个用于生成实体类 (C#) 的 T4 模板会自动创建出来。

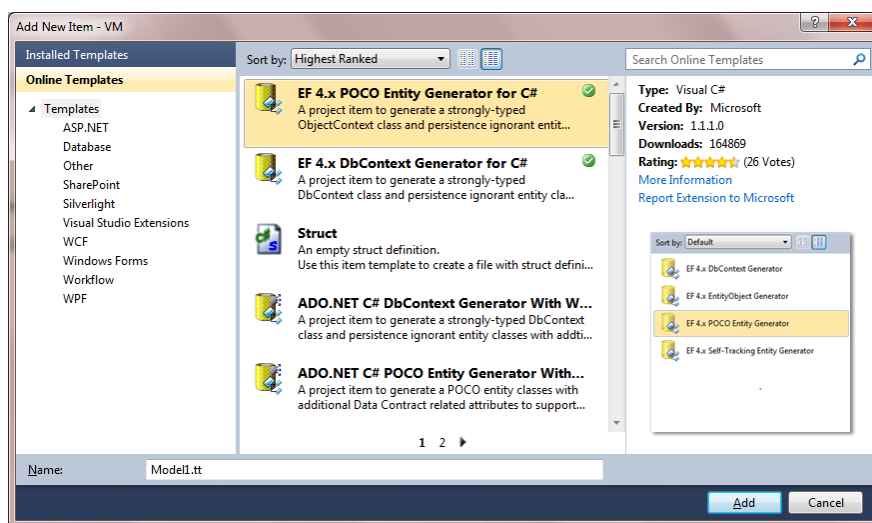


图 10-12 添加 POCO 实体类代码生成器

上面添加 T4 模板（一个扩展名为.tt 的文本文件）会读取实体数据模型的内容并为三个实体（Product、Order 和 OrderLine）生成具有如下定义的类，可以看到它们仅仅是单纯的 POCO 类型，既没有继承某个基类，类型和属性上也不曾应用任何特性。

```
public partial class Product
{
    public Product()
    {
        this.OrderLines = new HashSet<OrderLine>();
    }

    public string ProductId { get; set; }
    public string Name { get; set; }
    public string Genre { get; set; }
    public string Starring { get; set; }
    public string SupportingActors { get; set; }
    public string Director { get; set; }
    public string ScriptWriter { get; set; }
    public string ProductionCountry { get; set; }
    public string ProductionCompany { get; set; }
    public int ReleaseYear { get; set; }
    public string Language { get; set; }
    public int RunTime { get; set; }
    public decimal Price { get; set; }
    public string Story { get; set; }
    public string Poster { get; set; }
    public int Stock { get; set; }
    public string StoryAbstract { get; set; }

    public virtual ICollection<OrderLine> OrderLines { get; set; }
}
```



```

    }

    public partial class Order
    {
        public Order()
        {
            this.OrderLines = new HashSet<OrderLine>();
        }

        public string OrderId { get; set; }
        public string UserName { get; set; }
        public System.DateTime OrderTime { get; set; }

        public virtual ICollection<OrderLine> OrderLines { get; set; }
    }

    public partial class OrderLine
    {
        public string OrderId { get; set; }
        public string ProductId { get; set; }
        public int Quantity { get; set; }

        public virtual Order Order { get; set; }
        public virtual Product Product { get; set; }
    }

```

ADO.NET Entity Framework 使用 `DbContext` 实现与数据库的交互，可以采用与生成实体类的相同的方式生成基于指定实体数据模型的 `DbContext` 类。`DbContext` 类的生成使用的是图 10-12 所示的第二个选项（EF 4.x POCO `DbContext` Generator for C#）。由于我们的模型比较简单，所以我们选择自行定义它，这就是具有如下定义的 `VmDbContext`（其实在上面介绍 `Repository` 的设计的时候已经给出过它的定义）。

```

public class VmDbContext: DbContext
{
    public VmDbContext()
        : base("name=VM")
    {
    }

    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderLine> OrderLines { get; set; }
    public DbSet<Product> Products { get; set; }
}

```

IProductRepository 与 ProductRepository

每个具体的 `Repository` 都具有相应的接口，实现具体业务逻辑的 `Service` 采用接口的方式来调用 `Repository` 以实现对数据存储的抽象和隔离，用于获取商品数据的 `ProductRepository` 和它实现的接口 `IProductRepository` 定义如下。

```

public interface IProductRepository
{
    IEnumerable<Product> GetProducts(int pageIndex, int pageSize,
        out int recordCount);
    IEnumerable<Product> GetProductsByGenre(string genre, int pageIndex,

```

```

        int pageSize, out int recordCount);
IEnumerable<Product> GetProductsByActor(string actor, int pageIndex,
    int pageSize, out int recordCount);
Product GetProduct(string productId);
}

public class ProductRepository : VmRepository<Product>, IProductRepository
{
    public ProductRepository(VmDbContext dbContext)
        : base(dbContext)
    { }

    public IEnumerable<Product> GetProducts(int pageIndex, int pageSize,
        out int recordCount)
    {
        recordCount = this.DbSet.Count();
        return this.Get(p => true, pageIndex, pageSize,
            p => p.ReleaseYear, false);
    }

    public Product GetProduct(string productId)
    {
        Guard.ArgumentNotNullOrEmpty(productId, "productId");
        return this.Get(p => p.ProductId == productId).FirstOrDefault();
    }

    public IEnumerable<Product> GetProductsByGenre(string genre, int pageIndex,
        int pageSize, out int recordCount)
    {
        Guard.ArgumentNotNullOrEmpty(genre, "genre");
        recordCount = this.Count(p => p.Genre.Contains(genre));
        return this.Get<int>(p => p.Genre.Contains(genre), pageIndex, pageSize,
            p => p.ReleaseYear, false);
    }

    public IEnumerable<Product> GetProductsByActor(string actor, int pageIndex,
        int pageSize, out int recordCount)
    {
        Guard.ArgumentNotNullOrEmpty(actor, "actor");
        recordCount = this.Count(p => p.Starring.Contains(actor) ||
            p.SupportingActors.Contains(actor));
        return this.Get<int>(p => p.Starring.Contains(actor) ||
            p.SupportingActors.Contains(actor), pageIndex, pageSize,
            p => p.ReleaseYear, false);
    }
}

```

GetProducts、GetProductsByGenre 和 GetProductsByActor 均返回一个 Product 对象列表，并且采用了分页的机制，pageSize 和 pageIndex 参数分别表示每页的大小和当前页码，而输出参数 recordCount 返回符合条件的所有记录数量（不是返回记录数）。GetProducts 方法返回包含所有商品的列表，而 GetProductsByGenre 和 GetProductsByActor 方法获取指定类型和演员的影片列表。最后一个方法 GetProduct 用于获取指定 ID 的某个商品。

ProductRepository 继承自 VmRepository<Product>，它对四个方法的实现仅仅涉及到对基类 Get 和 Count 方法的调用，至于自定义 Guard.ArgumentNotNullOrEmpty 方法的调用，仅仅是确保字符串参数不为空（Null 或者 Empty）而已。

IOrderRepository 和 OrderRepository

OrderRepository 仅仅用于添加订单数据而已，所以我们只定义了一个参数类型为 Order 的 AddOrder 方法。OrderRepository 和它实现的接口 IOrderRepository 定义如下。

```
public interface IOrderRepository
{
    void AddOrder(Order order);
}

public class OrderRepository : VmRepository<Order>, IOrderRepository
{
    public OrderRepository(VmDbContext dbContext)
        : base(dbContext)
    { }

    public void AddOrder(Order order)
    {
        Guard.ArgumentNotNull(order, "order");
        this.Add(order);
    }
}
```

10.3.3 Service

我们直接将业务逻辑定义在 Service 中，每个具体的 Service 都具有对应的接口，服务的消费根据接口来完成。业务处理涉及的数据存取操作均通过相应的 Repository 来完成，而针对 Repository 的调用依然是通过接口的方式来进行的。

IProductService/ProductService

由于 VM 并没有涉及复杂的业务逻辑，所以除了多了一个用于获取指定商品库存量的 GetStock 方法之外，ProductService 的操作定义与 ProductRepository 基本上是一致的。ProductService 及其实现的接口 IProductService 的定义如下。

```
public interface IProductService
{
    IEnumerable<Product> GetMovies(int pageIndex, int pageSize,
        out int recordCount);
    IEnumerable<Product> GetMoviesByGenre(string genre, int pageIndex,
        int pageSize, out int recordCount);
    IEnumerable<Product> GetMoviesByActor(string actor, int pageIndex,
        int pageSize, out int recordCount);

    Product GetMovie(string productId);
    int GetStock(string productId);
}

public class ProductService : ServiceBase, IProductService
{
    public IProductRepository ProductRepository { get; private set; }
    public ProductService(IProductRepository productRepository)
```

```

    {
        this.ProductRepository = productRepository;
        this.AddDisposableObject(productRepository);
    }

    public IEnumerable<Product> GetMovies(int pageIndex, int pageSize,
        out int recordCount)
    {
        return this.ProductRepository.GetProducts(pageIndex, pageSize,
            out recordCount);
    }
    public IEnumerable<Product> GetMoviesByGenre(string genre, int pageIndex,
        int pageSize, out int recordCount)
    {
        Guard.ArgumentNotNullOrEmpty(genre, "genre");
        return this.ProductRepository.GetProductsByGenre(genre, pageIndex,
            pageSize, out recordCount);
    }
    public IEnumerable<Product> GetMoviesByActor(string actor, int pageIndex,
        int pageSize, out int recordCount)
    {
        Guard.ArgumentNotNullOrEmpty(actor, "actor");
        return this.ProductRepository.GetProductsByActor(actor, pageIndex,
            pageSize, out recordCount);
    }

    public Product GetMovie(string productId)
    {
        Guard.ArgumentNotNullOrEmpty(productId, "productId");
        return this.ProductRepository.GetProduct(productId);
    }
    public int GetStock(string productId)
    {
        Guard.ArgumentNotNullOrEmpty(productId, "productId");
        return this.ProductRepository.GetProduct(productId).Stock;
    }
}

```

通过前面的介绍我们知道 **Service** 和 **Repository** 都是采用 **IoC** 的方式在 **Controller** 被激活的时候创建的，通过上面的代码可以看出，**ProductRepository** 是以构造器注入的方式被初始化的。**ProductService** 继承自 **ServiceBase**，并在构造函数中通过调用 **AddDisposableObject** 方法将 **Repository** 对象添加到 **DisposableObjects** 集合中，使之能够在自身的 **Dispose** 方法被调用的时候被及时释放。

IOrderService/OrderService

如下所示的是实现订单提交的 **OrderService** 及其实现的 **IOrderService** 接口的定义，它具有唯一的服务操作 **SubmitOrder** 用于提交订购订单。**OrderService** 采用与 **ProductService** 一样的方式实现对 **OrderRepository** 的注入，并利用后者将提交的订单存储到数据库中。

```

public interface IOrderService
{
    void SubmitOrder(Order order);
}

```

```

public class OrderService : ServiceBase, IOrderService, IDisposable
{
    public IOrderRepository OrderRepository { get; private set; }
    public IProductService ProductService { get; private set; }

    public OrderService(IOrderRepository orderRepository,
        IProductService productService)
    {
        this.OrderRepository = orderRepository;
        this.ProductService = productService;

        this.AddDisposableObject(orderRepository);
        this.AddDisposableObject(productService);
    }

    [TransactionCallHandler]
    public void SubmitOrder(Order order)
    {
        Guard.ArgumentNotNull(order, "order");
        CheckStock(order);
        this.OrderRepository.AddOrder(order);
    }

    private void CheckStock(Order order)
    {
        foreach (var line in order.OrderLines)
        {
            if (this.ProductService.GetStock(line.ProductId) < line.Quantity)
            {
                throw new OutOfStockException("Out of stock...");
            }
        }
    }
}

```

我们在订单提交的时候添加了针对库存量的验证，而获取指定商品库存量的操作定义在 `ProductService` 中，所以我们定义了类型为 `IProductService` 的属性 `ProductService`，并和 `OrderRepository` 一样在构造函数中被初始化（构造器注入）。为了确保 `ProductService` 对象能够被正常地释放，我们依然通过调用 `AddDisposableObject` 方法将 `Repository` 对象添加到 `DisposableObjects` 集合中。

由于订单提交涉及到针对数据库的记录添加操作，我们需要通过事务确保其数据的完整性。在这里直接在 `SubmitOrder` 方法上面应用了我们前面定义的 `TransactionCallHandlerAttribute` 特性，如果通过 `Unity` 容器创建的 `OrderService` 对象支持基于 `Policy Injection` 的拦截机制，整个方法执行将会在一个 `TransactionScope` 中进行。

Unity 配置

VM 采用了 `Controller-Service-Repository` 这样的层次设计，相邻两个层次之间的调用均是利用接口来完成的，所以我们需要通过配置注册接口与实现类型之间的映射关系。由于 `OrderService` 利用了 `TransactionCallHandlerAttribute` 实现对事务的控制，我们需要在对其进行类型注册的时候进行 `Interception` 扩展的相关设置。如下所示的就是 VM 中 `Unity` 的完整配置。

```

<configuration>
  ...
  <unity>
    <sectionExtension
      type="Microsoft.Practices.Unity.InterceptionExtension.Configuration
        .InterceptionConfigurationExtension,
        Microsoft.Practices.Unity.Interception.Configuration"/>
    <containers>
      <container>
        <extension type="Interception" />
        <register type="Artech.VM.Services.IProductService, Artech.VM"
          mapTo="Artech.VM.Services.ProductService, Artech.VM"/>
        <register type="Artech.VM.Repositories.IOrderRepository, Artech.VM"
          mapTo="Artech.VM.Repositories.OrderRepository, Artech.VM"/>

        <register type="Artech.VM.Repositories.IProductRepository, Artech.VM"
          mapTo="Artech.VM.Repositories.ProductRepository, Artech.VM"/>
        <register type="Artech.VM.Services.IOrderService, Artech.VM"
          mapTo="Artech.VM.Services.OrderService, Artech.VM">
          <interceptor type="InterfaceInterceptor"/>
          <policyInjection />
        </register>
      </container>
    </containers>
  </unity>
</configuration>

```

10.3.4 路由注册和布局

我们可以注册相关的路由将 URL 设计得尽可能简洁并且具有指示性（用户通过 URL 就能知道请求何种资源或者执行何种操作），最重要的是使我们可以根据 SEO 的优化规则来对 URL 进行设计。通过本章开始提供的功能性介绍，我们知道 VM 的每一个页面的 URL 都经过精心的设计，而这体现在如下所示的路由注册代码中。

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        //主页[影片列表(第1页)]: /
        routes.MapRoute(
            name: "Home",
            url: "",
            defaults: new { controller = "Product", action = "Index",
                pageIndex = 1 } );

        //主页[影片列表(第N页)]: /Page1、/Page2、...
        routes.MapRoute(
            name: "Page",
            url: "Page{pageIndex}",
            defaults: new { controller = "Product", action = "Index",
                pageIndex = 1 },
            constraints: new { pageIndex = @"\d+" } );
    }
}

```

```

//基于指定类型的影片列表(第1页): /Genre/剧情、/Genre/喜剧、...
routes.MapRoute(
    name: "GenreHome",
    url: "Genre/{genre}",
    defaults: new { controller = "Product", action = "Genre",
        pageIndex = 1 });

//基于指定类型的影片列表(第N页): /Genre/剧情/Page1、/Genre/剧情/Page2、...
routes.MapRoute(
    name: "GenrePage",
    url: "Genre/{genre}/Page{pageIndex}",
    defaults: new { controller = "Product", action = "Genre",
        pageIndex = 1 }
);

//由指定演员参演的影片列表(第1页): /Actor/阿尔·帕西诺
routes.MapRoute(
    name: "ActorHome",
    url: "Actor/{actor}",
    defaults: new { controller = "Product", action = "Actor",
        pageIndex = 1 });

//由指定演员参演的影片列表(第N页): /Actor/阿尔·帕西诺/Page1、
//                                     /Actor/阿尔·帕西诺/Page2
routes.MapRoute(
    name: "ActorPage",
    url: "Actor/{actor}/Page{pageIndex}",
    defaults: new { controller = "Product", action = "Actor",
        pageIndex = 1 });

//影片详细信息: /魔鬼代言人/006
routes.MapRoute(
    name: "ProductDetail",
    url: "{productName}/{productId}",
    defaults: new { controller = "Product", action = "Detail" },
    constraints: new { httpMethod= new HttpMethodConstraint("GET") });

//购物车: /ShoppingCart
routes.MapRoute(
    name: "ShoppingCart",
    url: "ShoppingCart",
    defaults: new { controller = "Order", action = "ShoppingCart" });

//从购物车中移除选购商品: /ShoppingCart/Remove
routes.MapRoute(
    name: "Remove",
    url: "ShoppingCart/Remove",
    defaults: new { controller = "Order", action = "Remove" },
    constraints: new { httpMethod= new HttpMethodConstraint("POST") });

//结账支付: /CheckOut
routes.MapRoute(
    name: "CheckOut",
    url: "CheckOut",
    defaults: new { controller = "Order", action = "CheckOut" });

```

```

//用户登录: /Login
routes.MapRoute(
    name: "Login",
    url: "Login",
    defaults: new { controller = "Account", action = "Login" });

//登出注销: /Logout
routes.MapRoute(
    name: "Logout",
    url: "Logout",
    defaults: new { controller = "Account", action = "Logout" });
}
}

```

所有页面具有相同的页眉和页脚，页眉/页脚的内容被定义在布局文件中，如下所示的是整个布局文件的定义。

```

<!DOCTYPE html>
<html>
    <head>
        <title>@ViewBag.Title</title>
        <link rel="stylesheet" href="~/Content/Site.css" />
    </head>
    <body>
        <div class="page">
            <div class="header">
                <ul class="nav">
                    <li class="nav first">
                        
                        @Html.RouteLink("主页", "Home", null,
                            new { @class = "iconText" })
                    </li>
                    <li class="nav">
                        
                        @if (this.Request.IsAuthenticated)
                        {
                            @Html.RouteLink("注销", "Logout", null,
                                new { @class = "iconText" })
                        }
                        else
                        {
                            @Html.RouteLink("登录", "Login",
                                new { returnUrl = Request.Url },
                                new { @class = "iconText" })
                        }
                    </li>
                    <li class="nav">
                        
                        @Html.RouteLink("购物车", "ShoppingCart", null,
                            new { @class = "iconText" })
                    </li>
                </ul>
            </div>
            <div class="main">@RenderBody()</div>
            <div class="footer">

```



```

        <span><a href="mailto:jiangjiannan@gmail.com" target="_blank">
            联系作者 |</a></span>
        <span><a href="http://www.cnblogs.com/artech" target="_blank">
            技术博客 |</a></span>
        <span><a href="http://www.weibo.com/artech" target="_blank">
            新浪微博</a></span>
        <span class="copyRight">|©2012 Artech</span>
    </div>
</div>
</body>
</html>

```

10.3.5 ProductController

VM 的核心功能之一就是分页的方式将用户选择的商品列表呈现出来，这包括主页中默认显示的所有影片列表，以及用户点击“类型”和“演员”链接得到的相关影片的列表。当用户选择某部影片的时候，VM 还会将该影片的信息显示出来。这些都是通过 ProductController 来实现的。不过在实现 ProductController 之前，我们需要为最终呈现在 View 中的数据（Model）定义相应的数据类型。

MovieInfo 与 GeneralMovieInfo

通过第 1 章“ASP.NET + MVC”的介绍我们知道，ASP.NET MVC 是基于 MVC 模式的变体 Model 2 来设计的，所谓的 Model 实际上表示呈现在 View 上的数据，即 View Model。而基于业务规则的领域模型或者业务实体（比如 Product 类型）往往不能直接作为 View Model 使用，中间一般需要经过一个转换的过程。

我们定义了一个 MovieInfo 类型表示影片的详细信息。如下面的代码片段所示，MovieInfo 的数据成员的定义与 Product 大体类似，我们定义了静态方法 FromProduct 实现了从 Product 对象到 MovieInfo 的转换。MovieInfo 的属性 Genre、Starring 和 SupportingActors 类型变成了 IEnumerable<string>，以便于针对某个类型/演员生成相应的链接。海报图片的 Poster 表示的是图片真正的路径。

```

public class MovieInfo
{
    public string ProductId { get; private set; }
    [DisplayName("片名")]
    public string Name { get; private set; }
    [DisplayName("类型")]
    public IEnumerable<string> Genre { get; private set; }
    [DisplayName("领衔主演")]
    public IEnumerable<string> Starring { get; private set; }
    [DisplayName("主演")]
    public IEnumerable<string> SupportingActors { get; private set; }
    [DisplayName("导演")]
    public string Director { get; private set; }
    [DisplayName("编剧")]

```

```

public string ScriptWriter { get; private set; }
[DisplayName("制片国家")]
public string ProductionCountry { get; private set; }
[DisplayName("制片公司")]
public string ProductionCompany { get; private set; }
[DisplayName("发行年份")]
public int ReleaseYear { get; private set; }
[DisplayName("对白")]
public string Language { get; private set; }
[DisplayName("片长")]
public int RunTime { get; private set; }
[DisplayName("价格")]
public decimal Price { get; private set; }
[DisplayName("剧情介绍")]
public string Story { get; private set; }
public string Poster { get; private set; }

public static MovieInfo FromProduct(Product product)
{
    return new MovieInfo
    {
        ProductId      = product.ProductId,
        Name            = product.Name,
        Genre           = product.Genre.Split('|'),
        Starring        = product.Starring.Split('|'),
        SupportingActors = product.SupportingActors.Split('|'),
        Director        = product.Director,
        ScriptWriter    = product.ScriptWriter,
        ProductionCountry = product.ProductionCountry,
        ProductionCompany = product.ProductionCompany,
        ReleaseYear     = product.ReleaseYear,
        Language        = product.Language,
        Poster          = string.Format("~/images/poster/{0}", product.Poster),
        RunTime         = product.RunTime,
        Price           = product.Price,
        Story           = product.Story
    };
}
}

```

MovieInfo 作为显示影片详细信息页面的 **Model**，而对于显示影片列表的 **Model** 类型则是具有如下定义的 **GeneralMovieInfo**，它不具有 **Product** 所有的数据成员。

```

public class GeneralMovieInfo
{
    public string ProductId { get; set; }
    [DisplayName("片名")]
    public string Name { get; private set; }
    [DisplayName("类型/流派")]
    public IEnumerable<string> Genre { get; private set; }
    [DisplayName("领衔主演")]
    public IEnumerable<string> Starring { get; private set; }
    [DisplayName("导演")]
    public string Director { get; set; }
    [DisplayName("发行年份")]
    public int ReleaseYear { get; set; }
    [DisplayName("对白")]

```

```

public string Language { get; set; }
[DisplayName("价格")]
public decimal Price { get; set; }
[DisplayName("剧情介绍")]
public string StoryAbstract { get; set; }
public string Poster { get; set; }

public static GeneralMovieInfo FromProduct(Product product)
{
    return new GeneralMovieInfo
    {
        ProductId      = product.ProductId,
        Name            = product.Name,
        Genre           = product.Genre.Split('|'),
        Starring        = product.Starring.Split('|'),
        Director        = product.Director,
        ReleaseYear     = product.ReleaseYear,
        Language        = product.Language,
        Poster          = string.Format("~/images/poster/{0}", product.Poster),
        Price           = product.Price,
        StoryAbstract    = product.StoryAbstract
    };
}
}

```

分页导航

不论是主页针对所有影片列表的显示，还是显示指定类型或者由指定演员参演的影片列表，整个列表都是以分页的形式演示的。在页面右上角和右下角均具有一个包含页码的导航条，用户点击页码链接后指定某页的内容会被呈现出来。

这个分页导航条的 HTML 是通过我们为 HtmlHelper 定义的扩展方法 PageLinks 生成的。如下面的代码片段所示，PageLinks 具有一个类型为 PagingInfo 参数，它表示辅助分页的基本信息，包括每页的大小（PageSize，通过配置进行设置）、总记录条数（RecordCount）、当前页（PageIndex）和总页数，另一个参数提供一个委托对象用于根据指定的页码得到对应的 URL。

```

public static class HtmlUrlHelperExtensions
{
    //其他成员
    public static MvcHtmlString PageLinks(this HtmlHelper html,
        PagingInfo pagingInfo, Func<int, string> pageUrlAccessor)
    {
        StringBuilder result = new StringBuilder();
        for (int i = 1; i <= pagingInfo.PageCount; i++)
        {
            TagBuilder tag = new TagBuilder("a");
            tag.MergeAttribute("href", pageUrlAccessor(i));
            tag.InnerHtml = i.ToString();
            if (i == pagingInfo.PageIndex)
            {
                tag.AddCssClass("selected");
            }
            result.Append(tag.ToString());
        }
    }
}

```

```

        return MvcHtmlString.Create(result.ToString());
    }
}

public class PagingInfo
{
    public static int PageSize
    {
        get { return int.Parse(ConfigurationManager.AppSettings["pageSize"]); }
    }
    public int RecordCount { get; set; }
    public int PageIndex { get; set; }
    public int PageCount
    {
        get { return (int)Math.Ceiling((decimal)RecordCount / PageSize); }
    }
}

```

类型/演员链接

一部影片可以分为多种类型（比如剧情+喜剧），由多个演员参演，所以表示类型的属性 **Genre**、领衔主演的属性 **Starring** 和主演的属性 **SupportingActors** 的类型都是 **IEnumerable<string>**。不论是在影片列表页面还是详细信息页面，它们都是以链接的形式进行呈现的，而这些链接是通过我们为 **UrlHelper** 定义的两个扩展方法实现的。

如下面的代码片段所示，扩展方法 **GenreLinks** 和 **ActorLinks** 分别用于根据指定的字符串列表生成基于类型和参演演员的链接，而链接的目标地址是利用 **UrlHelper** 的 **RouteUrl** 方法根据注册的路由（“**GenreHome**”和“**ActorHome**”）生成的。

```

public static class HtmlUrlHelperExtensions
{
    public static MvcHtmlString GenreLinks(this UrlHelper helper,
        IEnumerable<string> genres)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string genre in genres)
        {
            sb.Append(string.Format("<span><a href=\"{0}\">{1}</a></span>",
                helper.RouteUrl("GenreHome", new { Genre = genre }), genre));
        }
        return new MvcHtmlString(sb.ToString());
    }

    public static MvcHtmlString ActorLinks(this UrlHelper helper,
        IEnumerable<string> actors)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string actor in actors)
        {
            sb.Append(string.Format("<span><a href=\"{0}\">{1}</a></span>",
                helper.RouteUrl("ActorHome", new { Actor = actor }), actor));
        }
        return new MvcHtmlString(sb.ToString());
    }
}

```

GeneralMovieInfo 模板

我们为作为表示影片列表项的 `GeneralMovieInfo` 类型定义了针对显示模式的模板（“~/Views/Shared/DisplayTemplates/GeneralMovieInfo.cshtml”）。如下面的代码片段所示，除了显示影片基本的信息之外，还会生成一个将当前商品添加到购物车的表单。

```
@model GeneralMovieInfo
<div class="product">
    @{
        MvcHtmlString htmlString = new MvcHtmlString(Model.StoryAbstract);
        string detailUrl = Url.RouteUrl("ProductDetail",
            new { Controller = "Product", Action = "Detail",
                ProductName = Model.Name, ProductId = Model.ProductId });
    }
    <a href="@detailUrl">
        
    </a>
    <div class="title">@Model.Name</div>
    <div>@Html.LabelFor(m => m.Genre): @Url.GenreLinks(Model.Genre)</div>
    <div>@Html.LabelFor(m => m.Starring): @Url.ActorLinks(Model.Starring)</div>
    <div>@Html.LabelFor(m => m.Director): @Model.Director</div>
    <div>@Html.LabelFor(m => m.Language): @Model.Language</div>
    <div>@Html.LabelFor(m => m.ReleaseYear): @Model.ReleaseYear</div>
    <div>@Html.LabelFor(m => m.StoryAbstract): @htmlString</div>
    <div class="addToShoppingCart">
        <form action="@Url.RouteUrl("ShoppingCart")" method="post">
            <span >@Html.LabelFor(m => m.Price):
                ¥@Model.Price.ToString("#0.00")
            </span>
            <input type="submit" value="" />
            <input type="hidden" name="ProductId" value="@Model.ProductId" />
            <input type="hidden" name="Price" value="@Model.Price" />
            <input type="hidden" name="ProductName" value="@Model.Name" />
        </form>
    </div>
</div>
```

影片列表 View

三种用于呈现影片列表的页面其实对应着同一个 `View`，这就是具有如下定义的 `MovieList`，这是一个 `Model` 为 `IEnumerable<GeneralMovieInfo>` 对象的强类型 `View`。它通过一个 `` 元素显示作为 `Model` 的 `GeneralMovieInfo` 列表，由于我们为 `GeneralMovieInfo` 类型定义的显示模板，所以只需要调用 `HtmlHelper<TModel>` 的 `DisplayFor` 扩展方法将 `GeneralMovieInfo` 对象以显示模式呈现出来即可。

```
@model IEnumerable<GeneralMovieInfo>
@{
    PagingInfo pagingInfo = new PagingInfo
    {
        PageIndex = (int)ViewBag.PageIndex,
        RecordCount = (int)ViewBag.RecordCount
    };
    Func<int, UrlHelper, string> pageUrlAccessor
```

```

        = (Func<int, UrlHelper, string>)ViewBag.PageUrlAccessor;
    }
    <div class="paging">
        @Html.PageLinks(pagingInfo, pageIndex=>pageUrlAccessor(pageIndex, Url))
    </div>

    <ul class="products">
        @foreach (GeneralMovieInfo movie in this.Model)
        {
            <li>
                @Html.DisplayFor(m => movie)
            </li>
        }
    </ul>

    <div class="paging">
        @Html.PageLinks(pagingInfo, pageIndex=>pageUrlAccessor(pageIndex, Url))
    </div>

```

View 在呈现 GeneralMovieInfo 列表的前后会调用我们为 HtmlHelper 定义的扩展方法 PageLinks 生成相应的分页导航条，服务分页的基本数据（当前页、记录总条数和根据页面获取 URL 的委托对象）都是通过 ViewBag 从 Controller 传递过来的。

ProductController

我们最后来介绍 ProductController 的定义。如下面的代码片段所示，它是 VmController 的子类，用于实现业务逻辑的 ProductService 属性（类型为 IProductService）在构造函数中被初始化（构造器注入）。在该构造函数中，我们调用 AddDisposableObject 方法将指定的 ProductService 对象置入 DisposableObjects 集合中使之能够随着 ProductController 的释放而释放。

```

public class ProductController : VmController
{
    public IProductService ProductService { get; private set; }
    public ProductController(IProductService productService)
    {
        this.ProductService = productService;
        this.AddDisposableObject(productService);
    }

    /// <summary>
    /// 显示所有影片列表（分页）
    /// </summary>
    public ActionResult Index(int pageIndex = 1)
    {
        int recordCount;
        IEnumerable<GeneralMovieInfo> movies = this.ProductService
            .GetMovies(pageIndex, PagingInfo.PageSize, out recordCount)
            .Select(p=>GeneralMovieInfo.FromProduct(p));
        Func<int, UrlHelper, string> pageUrlAccessor =
            (currentPage, helper) => helper.RouteUrl("Page",
                new { PageIndex = currentPage }).ToString();
        ViewBag.Title = "Video Mall";
        return RenderMovieList(movies, recordCount, pageIndex, pageUrlAccessor);
    }
}

```

```

/// <summary>
/// 显示由指定演员参演的影片列表 (分页)
/// </summary>
public ActionResult Actor(string actor, int pageIndex = 1)
{
    int recordCount;
    IEnumerable<GeneralMovieInfo> movies = this.ProductService
        .GetMoviesByActor(actor, pageIndex, PagingInfo.PageSize,
            out recordCount).Select(p=>GeneralMovieInfo.FromProduct(p));
    Func<int, UrlHelper, string> pageUrlAccessor = (currentPage,
        helper) => helper.RouteUrl("ActorPage",
            new { PageIndex = currentPage }).ToString();
    ViewBag.Title = actor;
    return RenderMovieList(movies, recordCount, pageIndex, pageUrlAccessor);
}

/// <summary>
/// 显示由指定类型的影片列表 (分页)
/// </summary>
public ActionResult Genre(string genre, int pageIndex = 1)
{
    int recordCount;
    IEnumerable<GeneralMovieInfo> movies = this.ProductService
        .GetMoviesByGenre(genre, pageIndex, PagingInfo.PageSize,
            out recordCount).Select(p => GeneralMovieInfo.FromProduct(p));
    Func<int, UrlHelper, string> pageUrlAccessor = (currentPage, helper) =>
        helper.RouteUrl("GenrePage",
            new { PageIndex = currentPage }).ToString();
    ViewBag.Title = genre;
    return RenderMovieList(movies, recordCount, pageIndex, pageUrlAccessor);
}

/// <summary>
/// 显示指定影片详细信息
/// </summary>
public ActionResult Detail(string productId)
{
    Product product = this.ProductService.GetMovie(productId);
    if (null == product)
    {
        return new HttpNotFoundResult(string.Format(
            "指定的产品\"{0}\"不存在", productId));
    }
    return View(MovieInfo.FromProduct(product));
}

private ActionResult RenderMovieList(IEnumerable<GeneralMovieInfo> movies,
    int recordCount, int pageIndex,
    Func<int, UrlHelper, string> pageUrlAccessor)
{
    ViewResult result = View("MovieList", movies);
    ViewBag.RecordCount = recordCount;
    ViewBag.PageIndex = pageIndex;
    ViewBag.PageUrlAccessor = pageUrlAccessor;
    return result;
}
}

```

三个 Action 方法 Index、Genre 和 Actor 用于获取对应影片列表,它们通过 ProductService

获取到相应的列表数据并作为 Model 被呈现在“MovieList”View 中。View 的最终呈现是通过私有方法 `RenderMovieList` 完成的，三个与分页相关的数据（记录总条数、当前页和根据页码获取对应 URL 的委托对象）通过 `ViewBag` 进行设置。

另一个 Action 方法 `Detail` 用于显示指定影片的详细信息，对应的 View 定义如下。这是一个 Model 为 `MovieInfo` 对象的强类型 View，在该 View 中除了将作为 Model 的 `MovieInfo` 对象的相关信息显示出来之外，还包含一个用于将当前商品添加到购物车的表单。

```
@model MovieInfo
@{
    ViewBag.Title = Model.Name;
}
<div class="productDetail">
    @{
        MvcHtmlString htmlString = new MvcHtmlString(Model.Story);
    }
    <img src=@Url.Content(Model.Poster) " class="poster" alt="@Model.Name" />
    <div class="title">@Model.Name</div>
    <div>@Html.LabelFor(m => m.Genre): @Url.GenreLinks(Model.Genre)</div>
    <div>@Html.LabelFor(m => m.Starring): @Url.ActorLinks(Model.Starring)</div>
    <div>@Html.LabelFor(m => m.SupportingActors):
        @Url.ActorLinks(Model.SupportingActors)</div>
    <div>@Html.LabelFor(m => m.Director): @Model.Director</div>
    <div>@Html.LabelFor(m => m.ScriptWriter): @Model.ScriptWriter</div>
    <div>@Html.LabelFor(m => m.Language): @Model.Language</div>
    <div>@Html.LabelFor(m => m.ProductionCountry): @Model.ProductionCountry</div>
    <div>@Html.LabelFor(m => m.ProductionCompany): @Model.ProductionCompany</div>
    <div>@Html.LabelFor(m => m.ReleaseYear): @Model.ReleaseYear</div>
    <div>@Html.LabelFor(m => m.Price): ¥@Model.Price.ToString("#0.00")</div>
    <div>@Html.LabelFor(m => m.Story): @htmlString</div>
    <div class="addToShoppingCart">
        <form action="@Url.RouteUrl("ShoppingCart")" method="post">
            <input type="submit" value="" />
            <input type="hidden" name="ProductId" value="@Model.ProductId" />
            <input type="hidden" name="Price" value="@Model.Price" />
            <input type="hidden" name="ProductName" value="@Model.Name" />
        </form>
    </div>
</div>
```

10.3.6 OrderController

`OrderController` 主要实现两个功能，一个是与购物车相关的，即显示购物车的商品列表和将选购的商品放入购物车；另一个则是根据购物车商品创建并提交订单。我们先来看看购物车在 VM 中是如何表示的。

ShoppingCart 与 ShoppingCartBinder

购物车和购物车中的商品条目通过具有如下定义的 `ShoppingCart` 和 `ShoppingCartItem` 表示。后者保存某个选购商品的 ID、名称、单价、属性和金额等信息，而 `ShoppingCart` 是一个 `ShoppingCartItem` 的列表（通过属性 `Items` 表示）。`ShoppingCart` 还具有两个计算类型的属

性 `TotalQuantity` 和 `TotalPrice`，它们分别表示购物车中商品总数量和总价。添加商品到购物车的功能实现在 `Add` 方法中，如果添加的商品不存在，则添加一个对应的 `ShoppingCartItem`，否则只是增加订购数量。

```
[Serializable]
public class ShoppingCart
{
    public IList<ShoppingCartItem> Items { get; private set; }

    public decimal TotalQuantity
    {
        get { return this.Items.Sum(item => item.Quantity); }
    }

    public decimal TotalPrice
    {
        get { return this.Items.Sum(item => item.Quantity * item.Price); }
    }

    public ShoppingCart()
    {
        Items = new List<ShoppingCartItem>();
    }

    public void Add(string productId, string productName, decimal price)
    {
        Guard.ArgumentNotNullOrEmpty(productId, "productId");
        Guard.ArgumentNotNullOrEmpty(productName, "productName");
        ShoppingCartItem shoppingCartItem = this.Items.FirstOrDefault(
            item => item.ProductId == productId);
        if (null != shoppingCartItem)
        {
            shoppingCartItem.Quantity++;
        }
        else
        {
            this.Items.Add(new ShoppingCartItem
            {
                ProductId = productId,
                ProductName = productName,
                Quantity = 1,
                Price = price
            });
        }
    }
}

[Serializable]
public class ShoppingCartItem
{
    public string ProductId { get; set; }
    [Display(Name = "片名")]
    public string ProductName { get; set; }
    [Display(Name = "单价")]
    public decimal Price { get; set; }
    [Display(Name = "数量")]
    public int Quantity { get; set; }
}
```

```

[Display(Name = "金额")]
public decimal SubTotalPrice
{
    get { return this.Quantity * this.Price; }
}
}

```

购物车的信息被保存在 `SessionState` 中。为了实现针对 `Action` 方法中 `ShoppingCart` 类型参数的自动绑定，我们特意为该类型定义相应的 `ModelBinder`——`ShoppingCartBinder`。如下面的代码片段所示，`ShoppingCartBinder` 的 `BindModel` 方法通过调用静态方法 `GetShoppingCart` 返回 `Model` 对象，而后者仅仅是获取保存在 `SessionState` 中的 `ShoppingCart` 对象而已。另一个静态方法 `Clear` 用于清除当前购物车。

```

public class ShoppingCartBinder: IModelBinder
{
    private const string KeyOfShoppingCart = "Artech.VM.ShoppingCart";
    public object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        return GetShoppingCart();
    }

    public static ShoppingCart GetShoppingCart()
    {
        if (HttpContext.Current.Session[KeyOfShoppingCart] == null)
        {
            HttpContext.Current.Session[KeyOfShoppingCart] = new ShoppingCart();
        }
        return (ShoppingCart)HttpContext.Current.Session[KeyOfShoppingCart];
    }

    public static void Clear()
    {
        HttpContext.Current.Session[KeyOfShoppingCart] = new ShoppingCart();
    }
}

```

OrderController

如下所示的是继承自 `VmController` 的 `OrderController` 的定义，真正实现了订单提交的 `OrderService` 属性（类型为 `IOrderService`）在构造函数中被初始化（体现了构造器注入）。为了使 `OrderService` 对象能够被正常释放，我们同样在构造函数中通过调用 `AddDisposableObject` 方法将指定的 `ProductService` 对象置于 `DisposableObjects` 列表中。

```

public class OrderController : VmController
{
    /// <summary>
    /// 实现了定购业务逻辑的 OrderService
    /// </summary>
    public IOrderService OrderService { get; private set; }
    public OrderController(IOrderService orderService)
    {
        this.OrderService = orderService;
        this.AddDisposableObject(orderService);
    }
}

```

```

    }

    /// <summary>
    /// 显示购物车商品列表
    /// </summary>
    public ActionResult ShoppingCart(
        [ModelBinder(typeof(ShoppingCartBinder))] ShoppingCart shoppingCart)
    {
        return View(shoppingCart);
    }

    /// <summary>
    /// 添加新的商品到购物车, 并显示当前购物车中商品列表
    /// </summary>
    [HttpPost]
    public ActionResult ShoppingCart(string productId, string productName,
        decimal price,
        [ModelBinder(typeof(ShoppingCartBinder))] ShoppingCart shoppingCart)
    {
        shoppingCart.Add(productId, productName, price);
        return View(ShoppingCartBinder.GetShoppingCart());
    }

    /// <summary>
    /// 删除购物车中某个商品
    /// </summary>
    [HttpPost]
    public ActionResult Remove(string productId,
        [ModelBinder(typeof(ShoppingCartBinder))] ShoppingCart shoppingCart)
    {
        ShoppingCartItem cartItem = shoppingCart.Items.FirstOrDefault(
            item => item.ProductId == productId);
        if (null != cartItem)
        {
            shoppingCart.Items.Remove(cartItem);
        }
        return RedirectToAction("ShoppingCart");
    }

    [Authorize]
    [HandleErrorAction("OnCheckOutError")]
    public ActionResult CheckOut(
        [ModelBinder(typeof(ShoppingCartBinder))] ShoppingCart shoppingCart)
    {
        Guard.ArgumentNotNull(shoppingCart, "shoppingCart");
        Order order = new Order
        {
            OrderId = Guid.NewGuid().ToString(),
            OrderTime = DateTime.Now,
            UserName = User.Identity.Name
        };
        foreach (ShoppingCartItem item in shoppingCart.Items)
        {
            order.OrderLines.Add(new OrderLine
            {
                Order = order,
                OrderId = order.OrderId,
                ProductId = item.ProductId,
            });
        }
    }

```

```

        Quantity    = item.Quantity
    });
}
this.OrderService.SubmitOrder(order);
ShoppingCartBinder.Clear();
return View();
}

/// <summary>
/// 用于处理从 Action 方法 CheckOut 抛出的异常
/// </summary>
public ActionResult OnCheckOutError(
    [ModelBinder(typeof(ShoppingCartBinder))]ShoppingCart shoppingCart)
{
    return View("ShoppingCart", shoppingCart);
}
}

```

针对 HTTP-GET 的 Action 方法 ShoppingCart 用于呈现购物车中商品列表，我们应用了 `HttpPostAttribute` 特性的 ShoppingCart 方法将指定的商品添加到购物车中。Action 方法 Remove 则用于从购物车中删除指定的商品。从上面的代码中可以看出，这些 Action 方法具有一个表示购物车的 ShoppingCart 类型的参数，该参数上应用了 `ModelBinderAttribute` 特性将对应的 `ModelBinder` 类型设置为 `ShoppingCartBinder`，这意味着该参数将会通过创建的 `ShoppingCartBinder` 进行自动绑定。

用于结账支付的 Action 方法 CheckOut 根据当前购物车的商品列表创建表示订购订单的 Order 对象，然后通过 OrderService 进行提交。值得注意的是该方法上应用 `HandlerErrorActionAttribute` 特性设置了发生异常时用于响应请求的另一个 Action，也就是说当在执行 CheckOut 方法过程中发生异常，另一个 Action 方法 OnCheckOutError 生成的 ActionResult 将最终用于响应请求。

由于结账支付请求是在购物车页面进行的，而 Action 方法 OnCheckOutError 显示的正是购物车页面，所以异常处理后的错误信息会自动呈现在购物车页面的 ValidationSummary 中。如图 10-13 所示，在库存不足的情况下提交订单，会在本页显示错误消息“库存不足”（错误消息是在配置文件中通过 ErrorMessageHandler 设置的）。

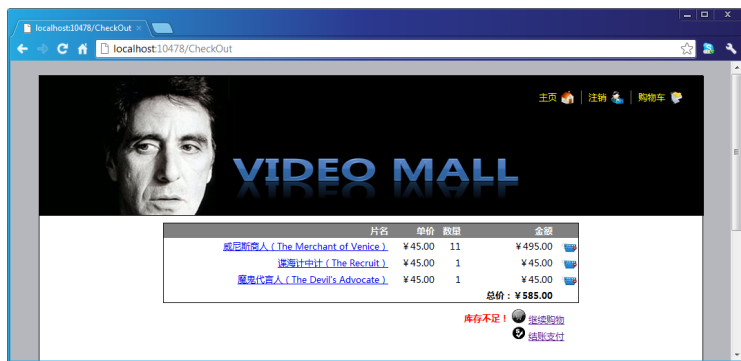


图 10-13 库存不足情况下错误消息的显示

如下所示的是代表购物车 View 的定义，这是一个 Model 为 ShoppingCart 对象的强类型 View，它遍历作为 Model 的 ShoppingCart 对象的选购商品列表，并将它们通过<table>的形式呈现出来。“继续购物”和“结账支付”两个链接被置于表格的右下方，用于显示错误消息的 ValidationSummary 就放在它们的左侧。

```
@model ShoppingCart
<div class="shoppingCart">
    <table cellpadding="0">
        <tr>
            <th>片名</th>
            <th>单价</th>
            <th>数量</th>
            <th>金额</th>
            <th></th>
        </tr>
        @foreach (ShoppingCartItem cartItem in this.Model.Items)
        {
            <tr>
                <td>@Html.RouteLink(@cartItem.ProductName, "ProductDetail",
                    new { ProductName = @cartItem.ProductName,
                        ProductId = @cartItem.ProductId })
                </td>
                <td>
                    ¥@cartItem.Price.ToString("#0.00")
                </td>
                <td>@cartItem.Quantity
                </td>
                <td>
                    ¥@cartItem.SubTotalPrice.ToString("#0.00")
                </td>
                <td>
                    <form action="@Url.RouteUrl("Remove")" method="post">
                        @Html.Hidden("productId", cartItem.ProductId)
                        <input type="submit" value="" class="remove" />
                    </form>
                </td>
            </tr>
        }
        <tr>
            <td colspan="3"></td>
            <td><b>总价: ¥@this.Model.TotalPrice.ToString("#0.00")</b></td>
            <td></td>
        </tr>
    </table>
    <div class="left">
        @Html.ValidationSummary()<div style="clear: both">
            </div>
        </div>
        <div class="right">
            <div>
                
                @Html.RouteLink("继续购物", "Home")
            </div>
            @if (ShoppingCartBinder.GetShoppingCart().TotalQuantity > 0)
```

```

    {
        <div>
            
            @Html.RouteLink("结账支付", "Checkout")
        </div>
    }
</div>
</div>

```

10.3.7 AccountController

AccountController 主要实现登录和注销功能，我们采用的是 Form 认证，而用户登录过程对用户的验证是借助 Membership 模块来完成的。具体来说，我们注册的 MembershipProvider 类型是 SqlMembershipProvider，如下所示的是相关的配置。

```

<configuration>
  <connectionStrings>
    <add name="VmDb"
        providerName="System.Data.SqlClient"
        connectionString="..." />
  </connectionStrings>
  <membership defaultProvider="SqlMembershipProvider">
    <providers>
      <add applicationName="VM"
          connectionStringName="VmDb"
          name="SqlMembershipProvider"
          type="System.Web.Security.SqlMembershipProvider,
              System.Web,
              Version=4.0.0.0,
              Culture=neutral,
              PublicKeyToken=b03f5f7f11d50a3a"
          requiresQuestionAndAnswer="false"
          minRequiredPasswordLength="6"
          minRequiredNonalphanumericCharacters="1" />
    </providers>
  </membership>
  <authentication mode="Forms">
    <forms loginUrl="~/Login" />
  </authentication>
</system.web>
</configuration>

```

由于我们并没有提供用户注册的功能，所以在 Global.asax 中编写了如下的代码使程序启动的时候创建一个默认测试账号（如果该账号不存在）。

```

public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //其他操作
        string userName = "Artech";
        string password = "Password+";

        if (Membership.FindUsersByName(userName).Count == 0)

```

```

        {
            Membership.CreateUser(userName, password, "artech@gmail.com");
        }
    }
}

```

如下所示的是整个 `AccountController` 的定义, 登录提供的用户名和密码封装在 `LoginInfo` 类型中。实现用户登录的 `Action` 方法 `Login` 在认证成功的情况下会重定向到之前的页面 (默认回到主页), 而 `Action` 方法 `Logout` 直接调用 `FormsAuthentication` 的静态方法 `SignOut` 完成用户注销, 注销之后会重定向到登录页面。

```

public class AccountController : VmController
{
    public ActionResult Login()
    {
        return View(new LoginInfo { UserName = "Artech" });
    }

    [HttpPost]
    [HandleErrorAction("OnLoginError")]
    public ActionResult Login(LoginInfo loginInfo, string returnUrl = "")
    {
        if (!ModelState.IsValid)
        {
            return View();
        }
        if (Membership.ValidateUser(loginInfo.UserName, loginInfo.Password))
        {
            FormsAuthentication.SetAuthCookie(loginInfo.UserName, false);
            if (string.IsNullOrEmpty(returnUrl))
            {
                return Redirect("/");
            }
            return Redirect(returnUrl);
        }
        else
        {
            //这段代码为了演示“自动化”异常处理
            if (Membership.FindUsersByName(loginInfo.UserName).Count == 0)
            {
                throw new InvalidUserNameException(
                    "Specified user account does not exists!");
            }
            throw new InvalidPasswordException(
                "Specified password is incorrect!");
        }
    }

    public ActionResult OnLoginError(LoginInfo loginInfo)
    {
        return View(loginInfo);
    }

    public ActionResult Logout()
    {

```

```

        FormsAuthentication.SignOut();
        string loginUrl = RouteTable.Routes.GetVirtualPath(
            ControllerContext.RequestContext, "Login", null).VirtualPath;
        return Redirect(loginUrl);
    }
}

public class LoginInfo
{
    [Required(ErrorMessage="请输入{0}")]
```

```

    [Display(Name = "用户名")]
    public string UserName { get; set; }

```

```

    [Required(ErrorMessage = "请输入{0}")]
```

```

    [Display(Name = "密码")]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}

```

为了演示自动化的异常处理,我们刻意在登录过程中抛出 `InvalidUserNameException` (输入用户名不存在) 和 `InvalidPasswordException` (输入密码不正确) 异常,异常处理配置中具有针对这两种异常类型的处理策略。该方法中同样通过应用的 `HandleErrorActionAttribute` 特性设置了对应的错误响应的 `Action` 方法 `OnLoginError`, 所以异常处理后的错误消息会以如图 10-14 所示的效果显示在当前页 (登录页面) 的 `ValidationSummary` 中。

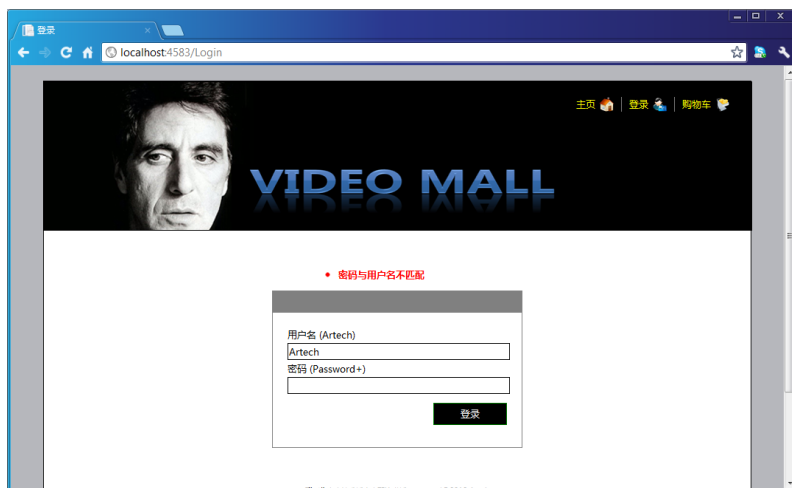


图 10-14 输入错误密码情况下错误消息的显示

如下所示的是作为登录页面的 `View` 的定义,这是一个 `Model` 为 `LoginInfo` 的强类型 `View`。

```

@model LoginInfo
@{
    ViewBag.Title = "登录";
}
<form action="@Url.RouteUrl("Login")" method="post">

```



```

<div class="login">
  <div>@Html.ValidationSummary()</div>
  <div class="title"></div>
  <div style="padding:20px; border:1px solid Gray; border-top-style:none">
    <div>@Html.LabelFor(m => m.UserName) <span>(Artech)</span></div>
    <div>
      @Html.TextBoxFor(m => m.UserName, new { @class = "textBox" })
    </div>
    <div>@Html.LabelFor(m => m.Password) <span>(Password+)</span></div>
    <div>
      @Html.PasswordFor(m => m.Password, new { @class = "textBox" })
    </div>
    <div>
      <input type="submit" value="登录" class="button" />
      @Html.Hidden("ReturnUrl",Request.QueryString["ReturnUrl"])
    </div>
  </div>
</div>
</form>

```

本章小结

我们创建了一个名为 Video Mal (VM) 的 Web 应用作为演示 ASP.NET MVC 的案例。VM 是一个在线销售影碟的电子商务网站,基本的功能包括商品列表的呈现、商品的订购和用户登录认证等。VM 使用 SQL Server 作为数据存储,并采用 ADO.NET Entity Framework 以 ORM 的形式进行数据库操作。

为了让 VM 对于真正的项目开发具有参考价值,模块化和层次化在 VM 得到了很好的体现。VM 的每个模块采用了 Controller-Service-Repository 三层结构,相邻的两个层次之间以及两个模块之间均通过接口进行交互,这样可以降低模块/层次之间的耦合度,同时也使得它们更加易于测试。

通过自定义的 ControllerFactory,我们实现了基于 IoC 的 Controller 激活方式,具体使用的 IoC 框架为 Unity。借助于 Unity 的 Interception 扩展,我们成功地将 AOP 引入到 VM 中。在自定义的 Controller 基类中,我们通过重写 OnException 方法实现了基于 EntLib 的自动化异常处理,并提供三种不同的异常消息响应方式。

附录 A 实例列表

第 1 章	S101	MVP (SC) 模式中 Presenter 与 View 之间的交互
	S102	按照 ASP.NET MVC 原理创建的模拟 MVC 框架
第 2 章	S201	通过 URL 路由实现请求地址与.aspx 页面的映射
	S202	基本路由注册
	S203	在路由注册中指定约束
	S204	忽略对于现有物理文件的路由
	S205	对于现有物理文件的路由
	S206	在路由注册中指定需要忽略的 URL 模式
	S207	通过注册的路由生成相应的 URL
	S208	注册路由映射与查看路由信息
	S209	一般 URL 参数与缺省 URL 参数
	S210	查看基于 Area 路由信息
	S211	创建一个 RouteHelper 模拟 UrlHelper 的 URL 生成逻辑
	S212	通过自定义 Route 对 ASP.NET 路由系统进行扩展
第 3 章	S301	Controller 默认的异步执行
	S302	通过 DisableAsyncSupport 属性实现 Controller 的同步执行
	S303	如何提升命名空间的优先级 (相同优先级命名空间下的多个同名 Controller 导致的异常)
	S304	如何提升命名空间的优先级 (为当前 ControllerBuilder 指定优先匹配命名空间)
	S305	如何提升命名空间的优先级 (在进行路由注册时指定优先匹配的命名空间)
	S306	Area 下的 Controller 的命名空间与对应 AdminAreaRegistration 不匹配导致的异常
	S307	移除 AdminAreaRegistration 的命名空间导致后备命名空间被使用
	S308	创建一个自定义 ControllerFactory 模拟 Controller 默认激活机制
	S309	IoC/DI 在 Unity 中的体现
	S310	创建基于 Unity 的 ControllerFactory
	S311	创建基于 Ninject 的 ControllerActivator
	S312	创建基于 Ninject 的 DependencyResolver

续表

第 4 章	S401	通过 UIHintAttribute 特性设置模板名称
	S402	通过 HiddenInputAttribute 特性设置“隐藏”元素 (1)
	S403	通过 HiddenInputAttribute 特性设置“隐藏”元素 (2)
	S404	通过 DataTypeAttribute/DisplayFormatAttribute 特性设置数据类型
	S405	通过 EditableAttribute/ReadOnlyAttribute 控制数据成员的读写性
	S406	通过 DisplayAttribute/DisplayNameAttribute 特性设置显示名称
	S407	通过 AllowHtmlAttribute 特性控制数据成员能否允许包含 HTML
	S408	创建实现 IMetadataAware 接口的特性定制 Model 元数据
	S409	通过模板将布尔值显示为 RadioButton
	S410	证明 DataTypeName 与模板名称的等效性
	S411	根据 Model 元数据获取“候选模板名称”列表
	S412	通过定制 Model 元数据和自定义模板实现预定义列表的呈现
	S413	通过自定义 ModelMetadataProvider 定制 Model 元数据
第 5 章	S501	返回 NameValueCollectionValueProvider 指定前缀的 Key (属性前缀)
	S502	返回 NameValueCollectionValueProvider 指定前缀的 Key (索引前缀)
	S503	探测 ChildActionValueProvider 的值提供机制
	S504	创建一个自定义 ValueProviderFactory
	S505	通过 ModelBinderAttribute 指定 ModelBinder 类型 (通过 Parameter Descriptor 获取 ModelBinder)
	S506	通过 ModelBinderAttribute 指定 ModelBinder 类型 (通过静态类型 Model Binders 获取 ModelBinder)
	S507	通过 ModelBinders 注册 ModelBinder
	S508	通过自定义 ModelBinderProvider 提供 ModelBinder
	S509	Model 绑定过程中对 ModelState 的设置
	S510	Model 绑定的默认实现 (简单类型)
	S511	Model 绑定的默认实现 (复杂类型 + 无前缀)
	S512	Model 绑定的默认实现 (复杂类型 + 有前缀)
	S513	Model 绑定的默认实现 (基于同名数据项的数组)
	S514	Model 绑定的默认实现 (基于零基索引的数组)
	S515	Model 绑定的默认实现 (基于文字索引的数组)
	S516	Model 绑定的默认实现 (集合)
	S517	Model 绑定的默认实现 (字典)

续表

第 6 章	S601	针对 DataErrorInfoModelValidator 的 Model 验证机制（在 Model 元数据中未指定被验证数据对象）
	S602	针对 DataErrorInfoModelValidator 的 Model 验证机制（在 Model 元数据中指定被验证数据对象）
	S603	CompositeModelValidator 采用的验证行为（ValidationAttribute 同时应用到容器类型和属性成员上）
	S604	CompositeModelValidator 采用的验证行为（ValidationAttribute 仅仅应用到容器类型上）
	S605	验证 Model 绑定过程中对 ModelError 的设置
	S606	调用 HtmlHelper 的扩展方法 ValidationMessage 显示验证消息
	S607	调用 HtmlHelper<Tmodel>的扩展方法 ValidationMessageFor 显示验证消息
	S608	调用 HtmlHelper<Tmodel>的扩展方法 ValidationSummary 显示验证消息
	S609	调用 HtmlHelper<Tmodel>的扩展方法 EditorForModel 自动显示验证消息
	S610	将 Model 验证实现在自定义的 ModelBinder 中
	S611	如何将多个同类 ValidationAttribute 特性应用到同一个目标元素上（未重写 TypeId 属性）
	S612	如何将多个同类 ValidationAttribute 特性应用到同一个目标元素上（重写了 TypeId 属性）
	S613	如何将 ValidationAttribute 特性应用到参数上
	S614	一种 Model 类型，多种验证规则
	S615	jQuery 验证（以内联的方式指定验证规则）
	S616	jQuery 验证（单独指定验证规则和错误消息）
	S617	自定义客户端验证
第 7 章	S701	异步 Action 的定义（XxxAsync/XxxCompleted）
	S702	异步 Action 的定义（Task 返回值）
	S703	ActionInvoker 的创建（未清空缓存）
	S704	ActionInvoker 的创建（清空缓存）
	S705	ActionInvoker 对 ControllerDescriptor 的创建
	S706	ReflectedAsyncControllerDescriptor 中的 ActionDescriptor 类型
	S707	AsyncController 和 ActionInvoker 对异步 Action 的影响
	S708	验证 Filter 的提供机制
	S709	验证 Filter 的执行顺序
	S710	分别应用到 Controller 和 Action 方法的筛选器（AllowMultiple 属性为 False）都有效吗
	S711	通过 ValidateInputAttribute 控制对包含 HTML 标签的输入数据的验证

续表

第 7 章	S712	设置 ActionExecutingContext 的 Result 对整个 ActionFilter 链执行流程的影响
	S713	集成 EntLib 实现自动化异常处理（将错误消息显示在当前页中）
	S714	集成 EntLib 实现自动化异常处理（将错误消息显示在错误页中）
	S715	集成 EntLib 实现自动化异常处理（针对 Ajax 的错误消息显示）
第 8 章	S801	不同情况下执行 Action 方法返回的 ActionResult
	S802	通过 ContentResult 实现主题定制
	S803	通过 FileResult 发布图片
	S804	利用 JavaScriptResult 实现动态 JavaScript 的执行
	S805	创建自定义 View
	S806	基于目录的 View 编译机制
	S807	创建一个简单的 RazorView
	S808	以 IoC 的方式激活 View
	S809	创建一个简单的 RazorViewEngine
第 9 章	S901	利用 Web API 结合 KO 实现简单的 CRUD 操作
	S902	HttpServer 的初始化与 HttpResponseMessage 管道的创建
	S903	自定义 HttpResponseMessage 实现 HTTP 方法重写
	S904	ControllerTypeResolver 对 HttpController 类型的解析
	S905	ControllerSelector 对 HttpController 的选择
	S906	通过自定义 HttpControllerActivator 实现基于 IoC 的 HttpController 的激活
	S907	通过自定义 DependencyResolver 实现基于 IoC 的 HttpController 的激活
	S908	Action 的 HTTP 方法支持规则（通过方法名决定支持的 HTTP 方法）
	S909	Action 的 HTTP 方法支持规则（通过相应的特性指定支持的 HTTP 方法）
	S910	HttpParameterBinding 的创建规则
	S911	针对控制台应用程序的 Web API 自我寄宿
第 10 章	S1001	通过 AOP 的方式实现针对方法返回值的缓存
	S1002	Vedio Mall

